Calhoun
Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1993

Modeling and simulation of a Deep Submergence Rescue Vehicle (DSRV) and its networked application

Zehner, Stanley Nelson.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/39861

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A271 742

DTIC
ELECTE
NOV 0 2 1993
S B D

THESIS

MODELING AND SIMULATION OF A
DEEP SUBMERGENCE RESCUE VEHICLE (DSRV)
AND ITS NETWORKED APPLICATION

by

Stanley N. Zehner

June 1993

Thesis Advisor: Dr. Robert B. McGhee
Co - Advisor: Dr. David R. Pratt

Approved for public release; distribution is unlimited.

93-26291

93 10 29 023

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): CS	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification): MODELING AND SIMULATION OF A DEEP SUBMERGENCE RESCUE VEHICLE (DSRV) AND ITS NETWORKED APPLICATION

12. PERSONAL AUTHOR(S): Zehner, Stanley N.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 09/91 TO 06/93	June 1993	142

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Deep Submergence Rescue Vehicle, Simulation, Physically based model, Real-time, Graphical model, Mathematical Model

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Computer real-time graphical simulations are in great demand. They save time, money and effort in the development of new hardware and training resources. Only recently have advances in computer hardware and software achieved a level which allow realistic simulations to run in real-time.

The phenomena we wish to simulate is increasingly complex. This in turn means that the software is becoming increasingly difficult to develop and maintain. The object oriented paradigm is one method of analysis and implementation which addresses the problems of increasing complexity.

This thesis examines the object oriented method by applying it to the problem domain of simulating the performance and handling characteristics of a U. S. Navy Deep Submergence Rescue Vehicle (DSRV). It performs an analysis of the key abstractions and implements the resulting design using the object oriented facilities of the C++ computer language.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Robert B. McGhee, Dr. David R. Pratt	22b. TELEPHONE (Include Area Code): (408) 656-2174; 22c. OFFICE SYMBOL: CS/M2

Approved for public release; distribution is unlimited

MODELING AND SIMULATION OF A DEEP SUBMERGENCE RESCUE VEHICLE (DSRV) AND ITS NETWORKED APPLICATION

by
Stanley Nelson Zehner
Lieutenant Commander, USN
Bachelor of Science, United States Naval Academy, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1993

Author: [redacted]
Stanley Nelson Zehner

Approved By: [redacted]
Dr. Robert B. McGhee , Thesis Advisor

[redacted]
Dr. David R. Pratt, Thesis Co - Advisor

[redacted]
Commander Gary J. Hughes, Chairman,
Department of Computer Science

ABSTRACT

Computer real-time graphical simulations are in great demand. They save time, money and effort in the development of new hardware and training resources. Only recently have advances in computer hardware and software achieved a level which allow realistic simulations to run in real-time.

The phenomena we wish to simulate is increasingly complex. This in turn means that the software is becoming increasingly difficult to develop and maintain. The object oriented paradigm is one method of analysis and implementation which addresses the problems of increasing complexity.

This thesis examines the object oriented method by applying it to the problem domain of simulating the performance and handling characteristics of a U. S. Navy Deep Submergence Rescue Vehicle (DSRV). It performs an analysis of the key abstractions and implements the resulting design using the object oriented facilities of the C++ computer language.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

TABLE OF CONTENTS

I. INTRODUCTION 1

A. MILITARY PROCUREMENT AND TRAINING 1

B. THE "SOFTWARE CRISIS" 2

C. GOALS 2

D. CHAPTER ORGANIZATION 2

II. CONCEPTUAL MODEL 4

A. THE DSRV MODEL 4

1. Kinematics 5

2. Mechanics And Rotational Dynamics 6

a. Rigid Body Components 6

b. Newton's Second Law Of Motion 7

c. Relevant Forces 8

B. THE SUBMARINE MODEL 10

C. SUMMARY 10

III. MATHEMATIC MODEL 11

A. FRAME OF REFERENCE CONVENTIONS 11

1. Engineering Covention 11

a. Global Reference Frame 11

b. Body Reference Frame 11

2. Graphics Hardware Convention 12

a. Global Reference Frame 12

b. Body Reference Frame 12

3. Reconciling The Conventions 12

B. THE DSRV MODEL 13

1. Kinematics 13

a. Linear Concepts 13
b. Rotational Concepts 14
c. Rotational Transformations 15
2. Mechanics And Rotational Dynamics 16
a. Vertical Forces: Gravity And Buoyancy 18
b. Hydrodynamic Forces 20
c. Control Forces 21
d. Ocean Current 23
3. Extended Form Of Vector Equations 23
4. DSRV Properites And Coeficients 25
C. THE SUBMARINE MODEL 25
D. SUMMARY 25
IV. OBJECT ORIENTED ANALYSIS AND DESIGN 27
A. THE OBJECT ORIENTED PARADIGM 27
B. OBJECT ORIENTED ANALYSIS VS. LANGUAGE 28
C. THINKING IN HIERARCHIES 29
D. PROBLEM ANALYSIS AND DESIGN 31
1. Levels Of Analysis And Granularity 31
2. Highest Level - The Simulator 31
3. Second Level - Internal Structure 32
4. Third Level - Fine Grain Objects 33
E. CLASS HIERARCHIES 33
1. DSRV And Submarine 34
2. Simulation Window 34
F. OBJECT HIERARCHIES 35
1. DSRV 35
2. Simulation Window 36
3. DSRV Simulator 37

G. SUMMARY 37
V. C++ OBJECT IMPLEMENTATION 38
A. AT THE TOP, THE MAIN() PROGRAM 38
B. HIGHEST LEVEL CLASS - DSRV_SIMULATION 38
C. THE DSRV_OBJ CLASS 39
D. THE SIMULATION_WINDOW CLASS 43
E. MATH MODEL APPLICATION 45
F. SUMMARY 45
VI. NETWORK ISSUES 46
A. NETWORK STUB 46
B. NETWORK ERROR THRESHOLDS 48
C. MULTIPLE CONCURRENT VIEWS 49
D. SUMMARY 49
VII. INTERFACE AND VISUALIZATION 50
A. OPERATOR INTERFACE 50
1. Spaceball 50
2. Keyboard 51
3. Menus 52
B. VISUALIZATION 53
C. SUMMARY 53
VIII. CONCLUSIONS AND RECOMMENDATIONS 54
A. CONCLUSIONS 54
B. RECOMMENDATIONS 54
1. Reuseable "Simulator" Base Class 54
2. Extend Default Class Functionality 54
3. Encapsulate Constraint Relationships In Classes 55
4. Interface And Display 55
5. Networked Virtual World 55

6. Better Numerical Integration Method 55
7. Collision Detection 56
C. FINAL REMARKS 56
List Of References 57
APPENDIX A 59
APPENDIX B 60
APPENDIX C 68
APPENDIX D 99
APPENDIX E 109
APPENDIX F 117
APPENDIX G 126
IInitial Distribution List.....131

LIST OF TABLES

TABLE 1 NEWTONIAN MECHANICS EQUATIONS. 17
TABLE 2 LONG FORM RIGID BODY EQUATIONS OF MOTION 24
TABLE 3 MAIN ANALYSIS MODELS. 29
TABLE 4 USER INTERFACE COMMANDS. 52

LIST OF FIGURES

Figure 1 . DSRV Thruster Arrangement 23

Figure 2 . The Simulator Abstraction. 32

Figure 3 . Second Level Simulator Object Abstraction 32

Figure 4 . Third Level Simulator Abstraction 34

Figure 5 . DSRV and Submarine Class Hierarchy 35

Figure 6 . Simulation Window Class Hierarchy 35

Figure 7 . DSRV Object Hierarchy 36

Figure 8 . Simulation Window Object Hierarchy. 36

Figure 9 . Simulation Object Hierarchy 37

Figure 10 Program main() 38

Figure 11 Class DSRV_Simulation. 39

Figure 12 Class Rigid Body 40

Figure 13 Class SubmarineVehicle 41

Figure 14 Class DSRV_obj, Public Members 42

Figure 15 Class DSRV_obj, Private Members 43

Figure 16 Class Simulation_Window. 44

Figure 17 VERN Network Class Hierarchy 47

Figure 18 Network Class Stubs. 48

Figure 19 SpaceBall Button Commands 51

Figure 20 DSRV Graphical Object From OFF File 53

I. INTRODUCTION

Converging trends build a compelling argument that computer graphical simulations will be central to future hardware procurement and systems development. Both the civilian and the military venue are affected. The trends driving this ascendancy are economic and technological; economic from dwindling resources and surging requirements, and technological from the forming critical mass between advances in computer hardware and the possibilities of new software paradigms. This work explores the possibilities offered by this critical mass and the ever increasing demand for computer simulations through the implementation of a real-time, networked, graphical simulation of a submerged Deep Submergence Rescue Vehicle (DSRV).

ThRee princip AL areas are addressed.

- Faithful simulation
- Object oriented analysis, design and implementation
- Networking interface

A. MILITARY PROCUREMENT AND TRAINING

Computer simulations certainly are not new. However, their recent marriage with high performance graphics workstations opens new possibilities for insight and efficiency. The military long encouraged computer driven trainers to mitigate the high cost of fielding tactical hardware in the training environment. Indeed, much of the early work in computer graphics originated from the early military aviation trainers [Rheingold91].

Today, the stakes are much higher than the periphery represented by hardware trainers. The government and industry are both trying to reduce costs without loss in effectiveness. Unending cost overruns in military procurement are no longer defensible. Large weapons systems no longer enjoy the luxury of being indispensable and, by extension, impervious

to the budget axe. There is strong support in the United States Congress to require a full modeled simulation of all future weapons acquisitions before prototyping and full scale production. Economic efficiency demands new methods. Those methods involve computers in general, and graphical simulations in particular.

B. THE "SOFTWARE CRISIS"

This term refers to the ever increasing complexity of software as we turn to software solutions for ever more difficult problems. Many tools are employed to manage complex software. The object oriented paradigm addresses the issues of software complexity and software maintenance. It is argued that a fundamental paradigm shift is needed to make the next quantum step in overcoming complexity is software systems.

C. GOALS

The goal of this thesis is to develop a realistic mathematical model of the handling characteristics of the DSRV and apply the model to a graphical, interactive simulation of the DSRV. In addition, it is desired to make the sumulation table driven (by coefficients of performance) for easy application to other submerged vessels and bodies.

Another goal is to develop the simulation within the framework of the object oriented analysis and design methodology and to implement the simulation using the object oriented capabilities of the C++ language. This approach is taken in the spirit of using these tools to alleviate the software crisis. These tools directly lend themselves to extensibility and code reuse.

D. CHAPTER ORGANIZATION

The intellectual activity of this project may be divided broadly into two categories: the underlying abstraction upon which the project rests, and the computer programming which implements the abstraction. This thesis reflects this structure. The first part deals with the

underlying abstraction. The second part deals with the programming implementation details.

This chapter outlines the motivations for this project, the perspective from which the project was undertaken, and a brief discussion of the organization of the thesis. The underlying abstraction is considered in Chapters II and III. Chapter II discusses the highest level of abstraction for the project, the conceptual model. It explains the conceptual basis of the DSRV and the underlying physics which makes up the physically based nature of the project. Chapter III extends the conceptual model into mathematical detail. It unifies three abstractions into the model:

- Newtonian kinematics,
- Newtonian dynamics, and
- Principles of naval architecture.

It explains the details of the mathematics and discusses the simplifying assumptions to make the model managable.

Chapter IV discusses the object oriented paradigm and how it was used in the analysis and design of this simulator. Chapter V completes the analysis and design by describing the implementation as it was conceived in the C++ language. Chapter VI discusses the network issues. It shows where the network interface is designed into the system and discusses some practical issues for the DSRV model.

Chapter VII covers the user interface and controls. It talks about the visual facilities and where they may be extended. Finally, Chapter VIII discusses the conclusions about this implementation and the areas where it might be extended.

II. CONCEPTUAL MODEL

The DSRV model is a comprehensive, physically based model which attempts to capture realistic handling and performance characteristics. Toward that goal, the model uses stability and handling characteristics derived by the Department of the Navy, Naval Ship Research and Development Center (formerly the David Taylor Research Laboratory) from DSRV model and vehicle testing.[NSRDC69]

The Newtonian force-based paradigm of motion is used in the DSRV model. Other paradigms could have been used; Hamiltonian, based on system energy (least action) [Finney90], Lagrangian, based on generalized or canonical coordinates for constrained motion problems, and others. However, the Newtonian paradigm is the basis of most ship stability information (curves of form), is well understood, is intuitive and carries well across applications. [Barzel92]

The DSRV mission presupposes a submarine which is in distress and in need of rescue. This application uses a submarine as a target vehicle. The submarine model is a simple model which differs from the full, more complex model used for the DSRV. It is not physically based as is the DSRV model, and derives its motion from a simple integration between current and commanded positions.

A. THE DSRV MODEL

Computer graphics animation relies heavily upon kinematic motion--motion which considers position and velocity, but does not consider mass and force [Barzel92]. Object motion in this case is a simple computation of incremental position changes over time from a known velocity. On the other hand, dynamics modeling considers mass and force relationships in object motion. From these relationships, new velocity values may be derived which then are used in the kinematic sense to determine incremental changes to

object position. This is termed *forward kinetics* and *forward dynamics* [Foley87] and is the framework for the DSRV model.

1. Kinematics

To elaborate, kinematics is not concerned with the causes of motion. It is strictly concerned with the relationships between position, velocity and acceleration as a function of time. It is often discussed using the analog of an abstract point which moves in a frame of reference but has no intrinsic properties of its own.

This abstract point moves in three dimensions. Therefore, convention holds that position, velocity and acceleration all may be represented in terms of components along the reference axes. Each axis component may itself be a parametric equation of time. In this case, the three component parametric equations are expressed in vector form and are termed vector-valued functions or simply *vector functions* of parameters and time [Finney90].

A kinematic rigid body is the next level of abstraction after the kinematic point. The kinematic rigid body expands the point abstractions of position, velocity and acceleration to include body orientation, body angular velocity and body angular acceleration. Descriptions of these values are conveniently expressed in terms of three dimensional components encapsulated in vector form. These components also may be expressed as parametric equations of time and, therefore, are vector functions.

In summary, the motion in three dimensions of a kinematic rigid body may be described in terms of vector functions of time which encapsulate the following body attributes:

- position
- velocity
- acceleration
- orientation
- angular velocity
- angular acceleration.

These vector functions may be integrated over time to yield their corresponding position and attitude values. These values then are used in the graphics function calls to display the object in three dimension space.

The next step is to define the parametric equations which make up the vector functions. For the DSRV model, the source is Newtonian mechanics and rotational dynamics.

2. Mechanics And Rotational Dynamics

The fundamental abstraction of the DSRV model is that of a submerged rigid body subjected to influencing forces and torques. This abstraction derives from the hierarchical development of Newton's laws of motion. The submerged rigid body derives from a rigid body, which derives from a point mass. At each level of abstraction more descriptions are added to define the relevant characteristics of the body. For the purposes of the conceptual model, all defining elements are assumed to be encapsulated into the concept of the rigid body.

a. Rigid Body Components

Mechanics and rotational dynamics deal with the physical concepts of mass, force, momentum and energy [Weidner75]. The one dimensional point mass abstraction is described as a mass and a velocity. Momentum is defined as the product of mass and velocity. Force is defined as the time rate of change of momentum, which may be rewritten as the product of mass and acceleration.

A point mass may move in a frame with three degrees of freedom; the three components of linear motion in three dimensions. In this case, the point mass is described as a scalar quantity, mass, and a directional velocity vector. Linear momentum is a vector quantity defined as the scalar multiplication of mass with the velocity vector. Linear force is a vector quantity defined as the time rate of change of linear momentum, which may be rewritten as the scalar multiplication of mass with an acceleration vector.

The rigid body abstraction adds another three degrees of freedom representing orientation. Orientation may be represented as a vector quantity of rotations about the frame axes needed to define an orientation. A whole series of definitions arise out of the rigid body abstraction which mirror their point mass counterparts. These definitions are, with their linear counterparts in parenthesis; moment of inertia (mass counterpart), angular velocity (velocity), angular momentum (linear momentum), and torque (force).

Summarizing, a non-minimal list of descriptive characteristics for a rigid body would include the following (vector values, unless otherwise specified):

- Mass (scalar)
- Scalar velocity (scalar)
- Position
- Velocity
- Acceleration
- Linear momentum
- Linear Force
- Orientation
- Angular velocity
- Angular acceleration
- Angular momentum
- Torque.

It should be noted that there are more derived properties of rigid bodies, such as work, kinetic energy and power. They certainly would be appropriate for further refinements of the conceptual model, especially for building constraint relationships and for extending the rigid body abstraction to articulated objects [Badler91]. These properties are not used in this analysis, however.

b. Newton's Second Law Of Motion

Newton's Second Law of Motion states that the time rate of change of an object's momentum is equal, in magnitude and direction, to the vector sum of *all* external forces acting on the object. This is the *superposition* principle of force, which means we may replace the vector sum of all acting forces with an equivalent, single resultant force.

This law is of supreme importance to the DSRV model and is the basis upon which all analysis and implementation rests.

It is important to note, also, that this law unites the two aspects of the physics involved in the model: the kinematics and the mechanics. If the forces acting on the rigid body are known and the proper accelerations are calculated (which will be the case for the DSRV model), then the velocity and displacement may be projected and used to update the graphical representation of the DSRV.[Weidner75]

For this model, the relevant forces act in six degrees of freedom, as discussed above. Therefore the model is concerned with both linear and rotational forces. All forces and torques will be resolved into a net force and a net torque which will be used to derive velocity, position, angular velocity and orientation.

c. Relevant Forces

The DSRV is modeled as a submerged rigid body. The relevant force are composed of those which arise from the nature of the submerged environment and those which arise from the inherent DSRV controls.

The environmental forces include the primary vertical forces, gravity and buoyancy. Gravity acts downward from the body center of gravity and is a function of mass distribution. Buoyancy acts upward from the body center of buoyancy and is a function of displaced volume. For marine vehicle stability, the center of gravity is below the center of buoyancy. These two vertical forces are not always colinear. When they are offset from each other they produce a moment arm which induces a torque and an attendant rotational motion. The forces are self correcting and will seek an equilibrium in which they are colinear and opposite in sign.

The hydrodynamic forces are simplified into an apparent drag force and an added mass term. Apparent drag is proportional to the square of the vehicle velocity relative to its medium, and to the cross-sectional area normal to the relative velocity vector. It is an opposing force with a direction vector opposite to the relative velocity vector.

Added mass terms account for the fluid reaction force of the surrounding water to the acceleration of the body through the medium. Added mass does not have the same value for all directions of acceleration[Healey92]. For complex bodies, such as the DSRV, these terms are difficult to compute. Consequently, the DSRV model uses a simplifying assumption that the DSRV added mass term is approximated by an appropriately dimensioned sphere.[Healey92]

Control forces include propulsion, control surfaces (planes) and thrusters. The DSRV uses a shrouded propeller with two degrees of freedom for primary thrust in cruising mode. This model approximates the shrouded propeller using the classic submarine propulsion arrangement of a propeller with one rotational degree of freedom, a rear mounted rudder for yaw control and a stern plane for pitch control. Like the shrouded propeller, the propeller/plane configuration operates at a displacement from the center of gravity. For any angles diverging from the longitudinal axis, the propulsor will introduce a moment arm and an attendant torque.

For hovering mode, the DSRV uses four thrusters; two forward and two aft. They are arranged so that they are parallel to the transverse axis and the vertical axis. When activated, they apply forces through a moment arm relative to the center of gravity and induce a torque. They are easily implemented in the DSRV model as forces acting through the appropriate moment arms.

Unlike the forces discussed so far, ocean current does not act as a force on a rigid body. Rather, it is the result of a velocity vector attached to the medium in which the submerged rigid body is operating. Indeed, it is noticeable only from the perspective provided by the world inertial reference frame. In the ocean reference frame, current does not exist. Consequently, ocean current is a vector subtraction from velocity terms which have been translated from body to world coordinates. It should be noted that the model does not account for accelerations induced by uneven pressure distributions associated with wave action and vortex effects.

Summarized below are the forces which have been discussed and which are considered relevant for the DSRV model. It certainly is not a complete treatment of all the forces and effects which contribute to the control and stability of the DSRV. However, within the constraints of computational efficiency, realism, and real-time, interactive simulation, it is considered sufficient to provide an acceptable degree of handling realism. The forces are summarized as follows:

- Vertical forces
  - Gravity
  - Buoyancy
- Hydrodynamic forces
  - Apparent drag
  - Added mass
- Control forces
  - Propulsion
  - Control surfaces
  - Thrusters
- Ocean current

B. THE SUBMARINE MODEL

The submarine model is a simple kinematic model which assumes a consistent response to throttle and rudder commands. Longitudinal velocity and yaw rate are each modeled as separate first order linear differential equations. Numerical integration is accomplished using the Heun integration method. An ideal autopilot is assumed to regulate velocity and yaw to a commanded value using an invariant time constant.[McGhee91]

C. SUMMARY

The DSRV is modeled using the Newtonian force based paradigm of motion. Relevant dynamic forces may be reduced to single force and torque components. From these components the kinematic problem may be solved and position and velocity of the body determined. The mathematical model is derived in the next chapter from the conceptual model discussed above.

III. MATHEMATIC MODEL

A. FRAME OF REFERENCE CONVENTIONS

Any detailed discussion of physical objects reacting in three dimensions must establish a frame convention. Unfortunately, there is no universal frame convention for the types of physical interactions this model represents. Indeed, many related disciplines adopt conventions which differ from one another. Two conflicting conventions are at work in this thesis. The graphics hardware of the Silicon Graphics (SGI) Reality Engine uses a convention of coordinate axes for object displays which differs from the standard convention used in most vehicle dynamics models; thus, there is a need to resolve this conflict.

1. Engineering Covention

a. Global Reference Frame

To consider the details of vehicle motion in the context of Newtonian physics, it suffices to define an arbitrary inertial reference frame with three orthogonal axes. This global frame establishes the reference by which the motions of interacting objects are measured. References then are made to body coordinate positions, attitudes and their derivatives.

b. Body Reference Frame

The standard body coordinate convention for aircraft and marine vehicles is defined as a right handed, three-dimension Euclidean space. The orthogonal coordinate axes are aligned relative to the vehicle body. The **X** axis is placed along the vehicle longitudinal axis, positive in the direction of the front of the vehicle. The **Y** axis is normal to the **X** axis and is positive to the right as seen looking along the **X** axis. For example, in

an aircraft the right wing would point in the positive **Y** direction. The **Z** axis is positive down, through the undercarriage of the vehicle. [Healey92]

2. Graphics Hardware Convention

a. Global Reference Frame

The Silicon Graphics Reality Engine is the hardware platform on which this model is implemented. The SGI uses the convention of a right handed, three-dimension Euclidean space. The orthogonal coordinate axes are aligned relative to the display screen. The horizontal axis is the **X** axis, positive from left to right. The vertical axis is the **Y** axis, positive from bottom to top. The **Z** axis is perpendicular to the screen, positive in the direction toward the viewer.

b. Body Reference Frame

The programmer defines body reference frames. However, the viewing graphics calls are relative to an eye point which is defined in global coordinates. Object location and orientation also is specified in terms of global transformations and rotations. In the interest of simplicity and code readability, therefore, the programmer is inclined to define a body frame consistent with the SGI global reference frame (i.e., body reference axes initially parallel to global reference axis).

3. Reconciling The Conventions

The mathematical model is derived using the standard engineering conventions discussed, above. All matrix transformations and rotations assume an orthogonal, global inertial reference frame which defines a world coordinate system. Similarly, all objects interacting within the global frame are assumed to have an orthogonal, local reference frame which defines a body coordinate system for the object.

Reconciling the reference conventions between the graphics hardware and the mathematical model occurs in the graphic system function calls. Appropriate sign

transformations are used to make the graphic motion directionally consistent with the mathematical model.

B. THE DSRV MODEL

We now consider frames of reference from a different perspective, Newtonian physics. Newtonian kinematics and dynamics are inseparable from the concept of frames of reference. We intuitively think in terms of reference frames when considering the relative motion of moving bodies. If two bodies move in parallel directions, to each other, they appear to be mutually stationary. To an outside observer in a different reference frame, both appear to be moving with respect to the observer's reference frame.

One only need consider the difference in perceived motion between a passenger in a moving car on a highway looking at another car moving at the same speed in the next lane, and the perception of the same scene by a pedestrian standing at the side of the road. To the passenger, the cars are relatively stationary. To the pedestrian, both cars are moving quite fast with respect to his own speed.

This difference in reference frames is critical to understanding the Newtonian physics underlying the DSRV model.

1. Kinematics

a. Linear Concepts

In kinematics, we model body motion based on a "fixed" reference frame. This frame is arbitrarily chosen and becomes the benchmark against which all further motion and kinematic relationships are measured. For the kinematic model, we call this reference frame the *world coordinate system* and define it to be an orthogonal, right-handed coordinate system with coordinate axes X, Y, and Z, and origin $(0, 0, 0)$.

Position of any point within the world coordinate system is expressed as the triple (X, Y, Z) which uniquely identifies the point. Considering unit vectors along each axis, position may be expressed as a vector of the three components (Eq 1).

$$\mathbf{R} = [X\mathbf{I} + Y\mathbf{J} + Z\mathbf{K}] \qquad \text{(Eq 1)}$$

By convention, this equation is often expressed in vector form using the scalar components of each unit vector (Eq 2).

$$\mathbf{R} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \qquad \text{(Eq 2)}$$

In a similar fashion, velocity of a point with respect to the world coordinate system may be expressed in terms of its components along the world coordinate axes (Eq 3).

$$\frac{d\mathbf{R}}{dt} = \begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} \qquad \text{(Eq 3)}$$

As discussed in Chapter II, the DSRV model is based upon the rigid body concept. The rigid body is an extension of the point mass abstraction and introduces the concept of orientation and angular derivatives (angular velocity and angular acceleration). This allows a model with six degrees of freedom.

b. Rotational Concepts

There are several methods useful for expressing orientation and angular derivatives. This model uses the Euler angle convention to express angular relationships. The use of quaternions is popular for this purpose, particularly in aircraft simulations [Cooke92]. Quaternions overcome a major limitation of Euler angles. Euler angles produce a singularity when one angle of the rotation reaches 90 degrees; for example, when an aircraft achieves vertical flight.[Healey92]

The DSRV does not achieve such steep trim angles in its operations so this drawback for Euler angles does not affect the simulation. Euler angles have the advantage of common use for such applications and are well understood. For these reasons the Euler angle convention is used in this simulation.

A new coordinate system is defined for a rigid body using the conventions discussed earlier. This system is termed the *body coordinate system*. Starting with all three body axes aligned with the world coordinate system, three successive rotations about the *body* axes define the orientation of the body. These rotations are the Euler angles *psi*, *theta* and *phi* (Ψ, θ, ϕ), about the body z, y, and x axes, respectively. They are commonly known as azimuth, elevation and spin[Healey92]. They uniquely define the orientation of the body with respect to the world coordinate system.

Focussing now on the body coordinate system, there are three angular rates which describe body angular motion but which are separate and distinct from Euler angles. These are yaw, pitch and roll rates, respectively. Rather than relating body orientation to world coordinates, these quantities are exclusively concerned with the inertial frame of the body. They are extremely important to rigid body dynamics. Rate gyros are installed on rigid bodies to measure *yaw rate*, *pitch rate* and *roll rate* (p, q, r). Through various transformations these quantities are transformed from values measured in body coordinates to Euler angle rates in world coordinates. The latter rates are in world coordinates and therefore can be integrated to update the body's position and orientation. These are the values which are used as arguments for the graphics system calls.

c. Rotational Transformations

The link between motion variables sensed in body coordinates to their corresponding equivalents in world coordinates is achieved through geometric transformations using Euler angles. The *forward* transformation is considered the transformation from world to body coordinates. The *reverse* transformation is considered the opposite, from body to world coordinates. The reverse transformation is most useful in this DSRV model and is derived from successive reverse transformations about the world coordinate axes using Euler angles (Eq 4).

$$\mathbf{T}^{-1}(\phi, \theta, \psi) = \mathbf{T}^{-1}(\psi)\,\mathbf{T}^{-1}(\theta)\,\mathbf{T}^{-1}(\phi) \qquad \text{(Eq 4)}$$

Completing the triple matrix product of the component transformation matrices yields the extended form of the reverse transformation matrix used in this model (Eq 5).

$$\mathbf{T}^{-1}(\phi, \theta, \psi) = \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi \\ \sin\psi\cos\theta & \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi \\ -\sin\theta & \cos\theta\cos\phi & \cos\theta\cos\phi \end{bmatrix} \quad \text{(Eq 5)}$$

The goal for kinematics in this model is to describe in world coordinate terms, the motion of the DSRV rigid body as measured in body coordinate terms. In particular, the simulation must apply these transformations to linear and angular velocity as seen in body coordinates. For linear velocities this is accomplished using the transformation matrix and applying it to body velocity terms (u, v, w) (Eq 6).

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \mathbf{T}^{-1}(\psi, \theta, \phi) \begin{bmatrix} u \\ v \\ w \end{bmatrix} \quad \text{(Eq 6)}$$

For angular velocities, a similar transformation is used to relate body angular rates (p, q, r) to world coordinate Euler angle rates. Shown below are the extended forms of these equations (Eq 7).

$$\begin{aligned} \dot{\psi} &= q\frac{(\sin\phi + r\sin\phi)}{\cos\theta} \\ \dot{\theta} &= q\cos\phi - r\sin\phi \\ \dot{\phi} &= p + q\sin\phi\tan\theta + r\cos\phi\tan\theta \end{aligned} \quad \text{(Eq 7)}$$

Together, the world coordinate linear and angular rate equations ((Eq 6) and (Eq 7)) form the basis for the kinematic model. These equations are integrated with respect to time using numerical integration methods to derive DSRV position and orientation values.

2. Mechanics And Rotational Dynamics

The model must establish a means of deriving body velocity values. It employs Newtonian physics to derive these values. The fundamental building blocks are

summarized in Table 1. They relate familiar translational concepts to their rotational counterparts[Healey92].

TABLE 1: NEWTONIAN MECHANICS EQUATIONS

Linear		Rotational	
Mass	m	Moment of Inertia	$\mathbf{I} = \sum m_i \mathbf{r}_i^2$
Velocity	v	Angular Velocity	ω
Linear Momentum	$\mathbf{p} = m\mathbf{v}$	Angular Momentum	$\mathbf{L} = \sum (\mathbf{r} \times m\mathbf{v})$
Acceleration	a	Angular Acceleration	α
Force	$\sum \mathbf{F} = m\mathbf{v}$	Torque	$\tau = \mathbf{r} \times \mathbf{F}$
Newton's 2nd Law	$\sum \mathbf{F} = \frac{d\mathbf{p}}{dt}$ $\sum \mathbf{F} = m\mathbf{a}$	Newton's 2nd Law	$\sum \tau = \frac{d\mathbf{L}}{dt}$ $\sum \tau = \mathbf{I}\alpha + \omega \times \mathbf{L}$

While most of the vector variables are fairly intuitive from their linear counterparts, the variables for moment of inertia are quite different. The use is analogous to the property of mass. However, while mass is a scalar value representing an intrinsic property of a particle and is independent of the motion of the particle, moments of inertia depends on both the object's mass and on the distribution of the mass relative to the axis of rotation [Weidner75]. For uneven distributions of mass or for an axis other than the symmetrical axes, the computations for moment of inertia can be complex.

The DSRV model makes several simplifying assumptions;

- the center of gravity is located at the origin of the axes of symmetry of the body (this assumption is only for calculations for moments of inertia),
- the body rotates about the symmetric axes, and
- the body is modeled as a solid cylinder, long dimension along the body x axis.

Typically the moments and products of inertia are arranged in a 3×3 matrix called the *inertial tensor matrix*. For our assumptions, the products of inertia are zero, and the moments of inertia - arrayed along the principal diagonal - are computed as shown in Equation (8). [Weidner75] [Badler91]

$$\mathbf{I} = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ I_{xy} & I_{yy} & -I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix} \qquad \mathbf{I} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & I_{yz} & 0 \end{bmatrix} \qquad \text{(Eq 8)}$$

$$I = \int r^2 dm = \int \rho r^2 dv$$

$$I_{xx} = \frac{1}{2} m r^2 \qquad I_{yy} = I_{zz} = \frac{1}{12} m l$$

where: m = DSRV mass

r = DSRV radius

Newton's second law, the superposition principle of force and torque, brings together all the applied forces and torques so that they may be treated as a single force and torque affecting the body.

Two, three-dimensional vector equations for force and torque form the basis for describing the translational and rotational motion of a rigid body [Badler91]. Equation (9) shows the relationships.

$$F = ma - m\rho_G \times \dot{\omega} + m\omega \times (\omega \times \rho_G)$$
$$\tau = \mathbf{I}\dot{\omega} + m\rho_G \times a + \omega \times \mathbf{I}\omega \qquad \text{(Eq 9)}$$

where ρ_G = vector to center

a. Vertical Forces: Gravity And Buoyancy

Gravity acts on a body, pulling it to the earth's center. Weight is the vertical force caused by gravity acting on a body in world coordinates. Its magnitude is the product of mass and gravity. The force is aggregated to act at a point in body coordinates. This point

is not normally co-located with the origin of the symmetric axes so it acts at a displacement from the body reference point.

Similarly, buoyancy acts at a displacement from the body reference point. It is a force proportional to the displaced volume of the body. For a submerged body in equilibrium, weight and buoyancy are colinear vectors of equal magnitude acting in opposite directions.

Weight and buoyancy are fixed in body coordinates. When the body undergoes attitude changes, weight and buoyancy are no longer colinear and therefore induce a righting moment about the metacenter (a term from naval architecture which is used as a reference point for describing interactions between weight, buoyancy). The righting moment causes the body to right itself by restoring the equilibrium of a colinear weight and buoyancy vector. [Comstock67]

Gravity and buoyancy act from locations in body coordinates, but their interactions are with respect to the world coordinate system (world "up" and world "down"). In world coordinates they act in the vertical axis so their components are strictly in the world Z axis. Equation (10) shows this relationships for weight and buoyancy.

$$W = 0I + 0J + (mg)K$$
$$B = 0I + 0J - (\rho V)K \qquad \text{(Eq 10)}$$

The world components then must be transformed to their components in body coordinates for use in the equations of motion. Equation (11) shows the total vertical force and moment vectors which are used in the equations of motion.[Healey92]

$$f_g = (W - B)\begin{bmatrix} -\sin\theta \\ \cos\theta\sin\phi \\ \cos\theta\cos\phi \end{bmatrix}$$

$$m_g = W\rho_G \times \begin{bmatrix} -\sin\theta \\ \cos\theta\sin\phi \\ \cos\theta\cos\phi \end{bmatrix} - B\rho_B \times \begin{bmatrix} -\sin\theta \\ \cos\theta\sin\phi \\ \cos\theta\cos\phi \end{bmatrix} \qquad \text{(Eq 11)}$$

where: ρ_G = vector to center of gravity from reference origin

b. Hydrodynamic Forces

There are several considerations for modeling hydrodynamic forces. The body may be either stationary or in motion. The fluid also may be either stationary or in motion. For the purposes of this simulation, the hydrodynamic forces are modeled for a body in motion in a stationary flow.

Hydrodynamic forces come from a modification of the pressure distribution around the surface area of the body. The modification is proportional to body velocity and acceleration. Two components make up the total hydrodynamic forces; drag and added mass.

Drag is proportional to vehicle velocity and is governed by the relationship in equation (12). The coefficient of drag varies with flow conditions, but is generally taken to be $C_d = 1.2$. The projected frontal area per unit length, D, is determined using the simplifying assumption that the frontal area is a sphere of radius equal to the width of the DSRV.

$$F_{drag} = \frac{\rho}{2} C_d D\,(u)\,|u| \qquad \text{(Eq 12)}$$

where ρ = sea water density

Added mass force is the effect of fluid mass that is accelerated in reaction to the normal acceleration of the vehicle. It is proportional to fluid density, the vehicle

projected area in the direction of motion, and vehicle acceleration. Equation (13)shows the relationship. The combined drag and added mass equation for linear motion is shown in Equation (14).

$$F_{am} = C_a \rho (\pi c^2) \frac{du}{dt} \qquad \text{(Eq 13)}$$

$$F_x = -F_{drag} - F_{am} \qquad \text{(Eq 14)}$$

c. Control Forces

Control forces are generated to provide some control of the vehicle as it operates in its designed medium. Control forces may come from many different installed motive generating equipment. Typically a submerged vehicle is outfitted with a propulsor of some kind and some combination of control planes and thrusters. Control over weight and buoyancy is typically maintained through the use of a ballast and trim system.

PROPULSION: The thrust force of a propeller is related to the following:

- water density, ρ,
- propeller diameter, D,
- speed of advance, V_a,
- gravity, g,
- rpm, n,
- fluid pressure, p, and
- fluid viscosity, μ.

The contribution of these factors may be determined experimentally and encapsulated into a nondimensional coefficient for thrust. Using the principle of similitude, the coefficient may be dimensionalized to the full scale model and applied to thrust calculations. The nondimensional coefficient equation for thrust is shown in equation (15). The dimensionalized coefficients and their relationship to thrust and the moment created by propeller thrust is shown in equations (16) and (17).[Comstock67][NSRDC67]

$$K_{thrust} = \frac{F_{propeller}}{\rho n^2 D^4} \qquad \text{(Eq 15)}$$

$$F_{propeller} = C_{thrust} n^2 \qquad \text{(Eq 16)}$$

$$M_{propeller} = x_{moment-arm} F_{propeller} \qquad \text{(Eq 17)}$$

CONTROL PLANES: As mentioned earlier, the DSRV does not have control plane surfaces. However, the shrouded propulsor, which has two degrees of freedom, behaves approximately as a screw-rudder-sternplane combination when operating in cruise mode. Using the rudder-sternplane model has the advantage of modeling the more traditional control surface configuration for submarines, making the model more easily extensible to other, more conventional submarine vehicles.

The only purpose of a control surface is to induce a moment on the vehicle to cause it to rotate and orient to a desired angle of attack. Besides the desired control forces, vortex shedding, drag forces and frictional forces are also induced. The total resultant forces may be resolved into a lift component (normal to the direction of the water stream incident to the plane surface), a drag component (parallel to the direction of motion of the water stream), and the desired control component, normal to the longitudinal axis of the vehicle. These forces may be expressed in nondimensional form so that tow tank test results may be extended to a full scale model.[*Comstock67*]

The DSRV model ignores the drag and lift forces induced by the rudder and models only the desired control force normal to the vehicle longitudinal axis. The control force is proportional to the water density, the cross-section plane area and the square of water velocity. The moment is the product of the control force and the displacement of the center of pressure of the control plane from the center of gravity. The control force and moment equations are shown in Equations (18) and (19), respectively.

$$F_y = \frac{C_n \rho A_T u^2}{2} \qquad \text{(Eq 18)}$$

$$M = F_y x \qquad \text{(Eq 19)}$$

THRUSTER: There are four thrusters on the DSRV, two longitudinal and two transverse as shown in Figure 1. They are used by the DSRV in hovering mode when

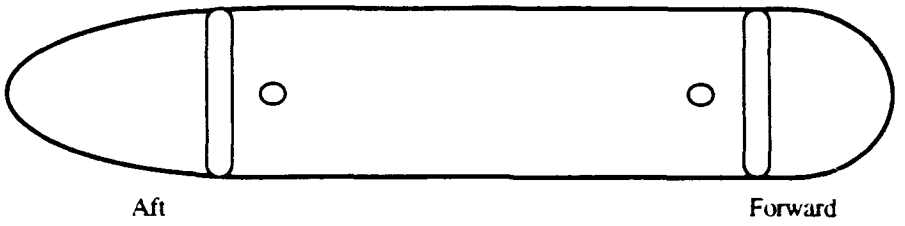

Figure 1 DSRV Thruster Arrangement

speeds are less than 2 knots. Thrusters are subject to the same factors as the propulsor, discussed above. They are modeled using the same equations but with their own corresponding coefficients.

d. Ocean Current

Ocean current acts with respect to the fixed world coordinate system. The DSRV maneuvers within the moving medium, so the effects of current are not sensed with respect to the vehicle body coordinate system. Therefore, current is an addition to the components of vehicle velocities which have been transformed to world coordinates (Eq 20).

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \mathbf{T}^{-1}(\phi, \theta, \psi) \begin{bmatrix} u \\ v \\ w \end{bmatrix} + \begin{bmatrix} U_{cx} \\ U_{cy} \\ U_{cx} \end{bmatrix} \qquad \text{(Eq 20)}$$

3. Extended Form Of Vector Equations

The DSRV mathematical model has been discussed in terms that invoke matrix shorthand notation. In the implementation, the model does not manipulate variables in vector form. Rather, the vector equations are extended to their long form and the components are calculated separately. Table 2 shows the long form of the equations of motion.[Healey92]

TABLE 2: LONG FORM RIGID BODY EQUATIONS OF MOTION

SURGE

$$X_f = m\,[\dot{u}_r - v_r r + w_r q - x_G(q^2 + r^2) + y_G(pq - \dot{r}) + z_G(pr + \dot{q})\,] + (W - B)\sin\theta$$

SWAY

$$Y_f = m\,[\dot{v}_r + u_r r - w_r p + x_G(pq + \dot{r}) - y_G(p + r) + z_G(qr - \dot{p})\,] - (W - B)\cos\theta\sin\phi$$

HEAVE

$$Z_f = m\,[\dot{w}_r - u_r q + v_r p + x_G(pr - \dot{q}) + y_G(qr + \dot{p}) - z_G(p^2 + q^2)\,] - (W - B)\cos\theta\cos\phi$$

ROLL

$$K_f = I_x\dot{p} + (I_z - I_y)\,qr + I_{xy}(pr - \dot{q}) - I_{yz}(q^2 - r^2) - I_{xz}(pq + \dot{r}) + m\,[y_G(\dot{w} - u_r q + v_r p) - z_G(\dot{v}_r + u_r r - w_r p)\,] - (y_G W - y_B B)\cos\theta\cos\phi + (z_G W - z_B B)\cos\theta\sin\phi$$

PITCH

$$M_f = I_y\dot{q} + (I_x - I_z)\,pr - I_{xy}(qr + \dot{p}) + I_{yz}(pq - \dot{r}) + I_{xz}(p^2 - r^2) - m\,[x_G(\dot{w}_r - \dot{u}_r q + v_r p) - z_G(\dot{u}_r - v_r r + w_r q)\,] + (x_G W - x_B B)\cos\theta\cos\phi + (z_G W - z_B B)\sin\theta$$

YAW

$$N_f = I_z\dot{r} + (I_y - I_x)\,pq - I_{xy}(p^2 - q^2) - I_{yz}(pr + \dot{q}) + I_{xz}(qr - \dot{p}) + m\,[x_G(\dot{v}_r + u_r r - w_r p) - y_G(\dot{u}_r - v_r r + w_r q)\,] - (x_G W - x_B B)\cos\theta\sin\phi - (y_G W - y_B B)\sin\theta$$

These equations do not explicitly present either added mass of control force and moment effects. Instead, these factors are all contained in the right hand side of each equation.

4. DSRV Properites And Coeficients

The concept of similitude has been mentioned several times with little explanation. Mechanical similitude is a well documented method in naval architecture for extrapolating ship model basin test results to the properties of full scale vehicles. Through a process of dimensional analysis, engineers are able to measure the forces and torques experienced by small models when moved and rotated through a tow tank. The experimental results are reduced to non-dimensional coefficients using various dimension normalizing techniques. Typically the dimensions are normalized for an appropriate power of vehicle length or area, depending on the computations and dimensional analysis. From nondimensional coefficients the full scale coefficients may be determined by applying the appropriate dimensional quantity from the full model (i.e.; length^1, length^2, length^3, etc.).[Comstock67]

C. THE SUBMARINE MODEL

The submarine model uses two simple equations to determine vehicle velocity and yaw rate (course). The equations model a perfect autopilot to regulate speed and course. The autopilot is assumed to regulate velocity and course changes using a fixed time constant, τ_a. Equations (21) and (22) govern the response of the submarine model.

$$\dot{u} = \frac{1}{\tau_a}(u_c - u) \qquad \text{(Eq 21)}$$

$$\dot{r} = \frac{1}{\tau_a}(r_c - r) \qquad \text{(Eq 22)}$$

Body velocities and angular rates are subjected to the same transformation from body to world coordinates as discussed for the DSRV model. Numerical integration is accomplished using the Heun integration method.[McGhee75]

D. SUMMARY

The equations of motion for a submerged rigid body define a mathematical model of the behavior of the DSRV. Factoring in the relevant control forces provides a realistic

model for characteristics. The model must be implemented in a programming language and tied into the graphics system calls to provide a real-time graphical simulation. The analysis and design of the software structure to implement the model is the subject of the next chapter.

IV. OBJECT ORIENTED ANALYSIS AND DESIGN

A. THE OBJECT ORIENTED PARADIGM

We speak of analysis, design and programming as distinct activities. In large, non-trivial software projects, these activities have been "codified" into a framework for dealing with complexity. Conventional structured analysis, as articulated by Yourdon [Yourdon89], is one example of such a framework. It is based upon the capabilities provided by common imperative languages, such as the language C. Booch, among others, discusses a framework based upon the object oriented paradigm [Booch91].

The driving force for articulating effective software development methods comes from the yet unresolved *software crisis*. Software is notoriously expensive, delivered late and often full of bugs. While development tools have been created over the years to address these problems, they have been met with new demands from software developers who are addressing ever increasingly complex problems. The tools also must become more complex to deal with the direction of software development.[Cantu92]

Booch states that software is inherently complex for four essential reasons:

- complexity of the problem domain,
- management of the development process,
- inherent flexibility of software, and
- the discrete nature of digital systems.

Complexity of the problem domain refers to the unending quest to apply computer solutions to higher level problems. Just as one level becomes achievable from computer advances, demands are made to extend computer solutions one abstraction higher.

Managing the development process addresses the ever increasing size of software. Computer solutions made possible by several hundred lines of code may be understood by a single programmer. Today's solutions run into the realm of a million lines of code which no single programmer can expect to fully grasp. Therefore, large programming applications

will take many programmers and, with turnover in employment, will likely finish with few if any of the original programmers still on the team.

The inherent flexibility of software stems from its ability to express virtually any kind of abstraction. This quality is a double edged sword because programmers are largely forced to construct all the component pieces of software with little or no standardization from one application and one programmer to another.

The discrete nature of digital systems refers to the fact that computer programs are a collection of discrete variable states. With large systems, the combinatorial explosion of these states makes it virtually impossible to guarantee that all state combinations may be tested. The testing requirements for a 100 percent tested system in any large software application would take several hundred or thousand years to completely test. Clearly, this situation is unacceptable and other methods must be used to ensure the veracity of the software.

So, the problem is to devise effective tools to deal with the issues of large software system development. The object oriented paradigm is such a tool. Unlike the classical formulation of a tool, the object oriented design philosophy is what is often referred to as a "paradigm shift"; a wholly new way of thinking about the problem domain and of designing software in the hope that this new intellectual approach will provide a framework for skirting the pitfalls of earlier software development processes.[Booch91]

B. OBJECT ORIENTED ANALYSIS VS. LANGUAGE

It is important to draw a distinction between the analysis method and the language which implements the design. The two are different, though related. The analysis method is the way in which the problem domain is analyzed and the solution is conceptualized. It is, itself, a high level conceptual model of analysis. Many analysis models have been employed. The five main types are summarized in Table 3.[Booch91]

The implementation language exists apart from the analysis style used. The language facilitates the implementation of the style. It must provide the facilities to implement the

TABLE 3: MAIN ANALYSIS MODELS

STYLE	ABSTRACTION
Procedure - oriented	Algorithms
Object - oriented	Classes and objects
Logic - oriented	Goals, predicate calculus
Rule - oriented	If - then rules
Constraint - oriented	Invariant relationships

style without too much trouble and without too many impediments to the concepts endemic to the style. The object model demands a language which provides four major elements. The absence of any one of the elements obviates the object oriented nature of the model. These elements are,

- Abstraction,
- Encapsulation,
- Modularity, and
- Hierarchy.

The C++ language provides facilities which support these elements. It should be noted that C++ also supports the procedure oriented paradigm. Therefore, an object oriented implementation must consciously seek out and use the object facilities of C++. The DSRV model uses the object model of analysis and the object oriented facilities of the C++ language to implement the model.

C. THINKING IN HIERARCHIES

According to Booch, humans deal with complexity by dividing abstract ideas into hierarchies. Hierarchies are simply an ordering of abstractions. Most areas of human intellectual analysis have been divided into hierarchies: the internal structure of the computer, the study of botany and zoology, the decomposition of physics into discrete concepts, and the nature of man's social institutions to name a few.[Booch91]

In contrast, the evolution of programming paradigms follows the sequential nature of the programs themselves: Functional decomposition, algorithmic structure, function calls,

loop statements, etc. The real promise of the object oriented paradigm is the unity between how humans think and the programming paradigm used to analyze and implement the abstractions of human thought.

It is no surprise, then, that the object model draws heavily upon hierarchical relationships. In particular, two interrelated conceptual hierarchies are used, the "kind of" hierarchy and the "part of" hierarchy. The "kind of" hierarchy refers to the *class structure* where a class is a description of the internal state and interface of an entity. This hierarchy is often thought of as a generalization/specialization hierarchy where superclasses are generalizations of subclasses which are specializations of an abstraction.

The "part of" hierarchy refers to the *object structure* where an object is an instance of a class (which may, itself, encapsulate other classes). In this hierarchy, the class is the higher level of abstraction compared to the encapsulated classes.

If hierarchies are central to the object model, then the programming language must have the facilities to support hierarchies. There are three fundamental facilities which must be present:

- Classes,
- Inheritance, and
- Polymorphism[Cantu92].

Classes allow abstractions to be encapsulated and modularized. Inheritance defines a relationship among classes which allows the class to use the structure or behavior of one or more other classes (for single and multiple inheritance, respectively). Booch states that programming without inheritance is "distinctly not object - oriented" and is termed programming with abstract data types. Polymorphism allows objects of different classes to use different member functions with the same name, where the classes are related by a common super class. Polymorphism is made possible in a language by the presence of the inheritance facility and dynamic binding.

D. PROBLEM ANALYSIS AND DESIGN

Analysis and design are discussed as distinct activities, but it is acknowledged that the distinction is often blurred and that they are more realistically considered as an uneven continuum of activity [Booch91]. In this project, analysis is considered to be the activity of identifying the key abstractions from the problem domain and the interactions among the abstractions. Analysis establishes an architecture for the key abstractions. It is the intellectual "construction" of a point-of-view which solidifies and gives context to the key abstractions.

The design activity is a "decomposition" phase in which the key abstractions of the problem domain are considered in relation to each other and in relation to abstractions *outside* the immediate problem domain. For example, in this application there is an object, a graphical simulation window, which displays the DSRV. Outside of the DSRV simulation problem domain, this window exists in the context of its own hierarchy of graphical windows and higher graphical objects. Placing this object in its correct context within a larger world domain is a proper activity for the design of object oriented systems.

1. Levels Of Analysis And Granularity

Analysis of the problem domain starts at the highest level of abstraction and successively considers the abstractions which support the higher levels. This top down approach is well understood and lends itself to the object oriented analysis approach. At successively deeper traversals of the abstraction tree, the granularity becomes finer. There are no hard and fast rules for defining the correct level of granularity. So, the analysis stops at an arbitrary level at which it appears the analysis arrives at an abstraction simple enough to need no further explanation.

2. Highest Level - The Simulator

The DSRV simulator exists by itself at the top of the abstract conceptual tree. It is the root node. It encompasses all the ideas, concepts and assumptions about simulating

the performance and creating the appearance of the DSRV. This simple abstraction is shown in Figure 2.

Figure 2 The Simulator Abstraction

3. Second Level - Internal Structure

The second level explores the structure of the simulator. The simulator must orchestrate the movement of the objects to be simulated. This involves controlling both the mathematical models and the graphical displays. The simulator also must be able to respond to commands from the operator. Thus, the second level of abstraction decomposes as shown in Figure 3.

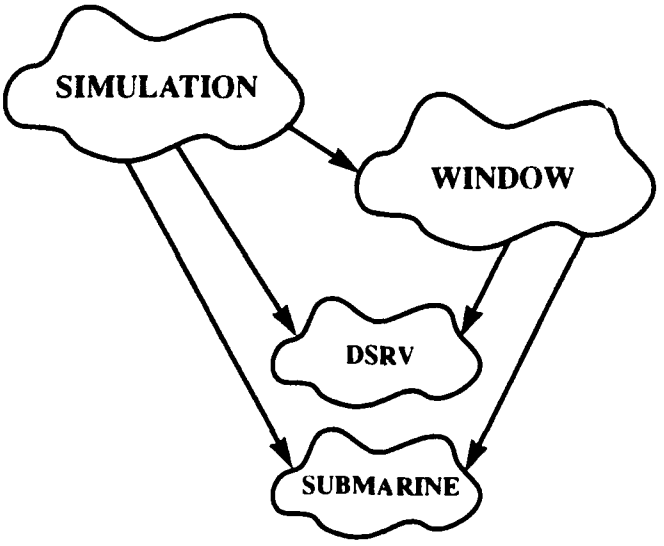

Figure 3 Second Level Simulator Object Abstraction

The Simulation entity controls the various functions necessary to conduct the simulation. These functions include timing the pace of the simulation, directing the display when the simulation is running and taking action on operator inputs when prompted. The Window abstraction is one of a controlling entity which orchestrates the various components of the display. It is responsible for window menus, background objects (such as sea bottom, light model and any non-moving objects) and the moving objects of the simulation after their positions have been updated. The vehicles are responsible for their own performance and handling characteristics. As they receive change instructions they must respond according to their defined nature.

Booch discusses a "using" relationship among objects. He generalizes three roles within this relationship:

- Actor - operates on other objects but is not operated upon,
- Server - never operates on other objects, only acted upon by others,
- Agent - both operates and is operated upon.

At this level, the Simulation is an actor, the Simulation Window is an agent, and the Simulated Vehicles are servers.

4. Third Level - Fine Grain Objects

This is the last level of decomposition in this problem domain.The Simulation and the individual vehicles are sufficiently described in the second level of the abstraction. The Window abstractions needs further definition. At this level, the lowest level, The Window abstraction introduces the view entity which represents different aspects and viewpoints into the virtual environment. Figure 4 shows this level of the analysis.

E. CLASS HIERARCHIES

As mentioned above, a class hierarchy represents a "kind of" relationship. In the DSRV model, three principal classes are analyzed in the context of their location in a class hierarchy, the DSRV, the Submarine and the Simulation Window.

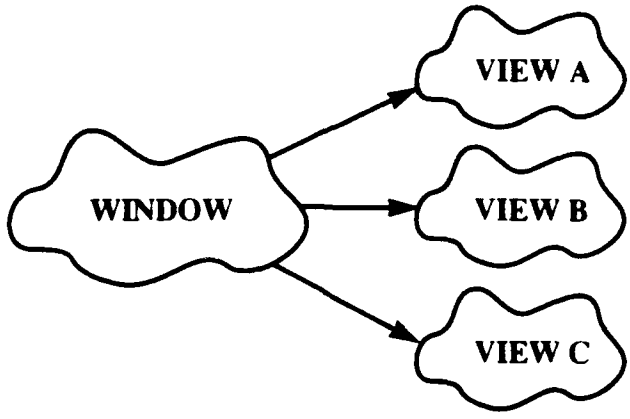

Figure 4 Third Level Simulator Abstraction

1. DSRV And Submarine

The DSRV and the Submarine both derive along the same class hierarchy. For this implementation, a pure physics hierarchy was considered first. The hierarchy follows the progression of classical Newtonian physics. Figure 5 shows the progression. Starting from the physics abstraction of a *point mass*, the hierarchy follows to succeeding derived sub - classes of *rigid body* and *submarine vehicle*. From submarine vehicle, the class *DSRV* and *Submarine* both derive to their own classes.

The *OFF Drawable Object* class is inherited by the Submarine Vehicle class. This allows all successive sub-classes to be used in a graphical environment.

2. Simulation Window

The *Simulation Window* class derives from the *Window* class. The Window super-class encapsulates the fundamental properties and behavior needed to define a graphical window. The Simulation window sub-class inherits these basic properties and provides additional data and functionality for providing simulation specific behavior. Figure 6 shows this relationship.

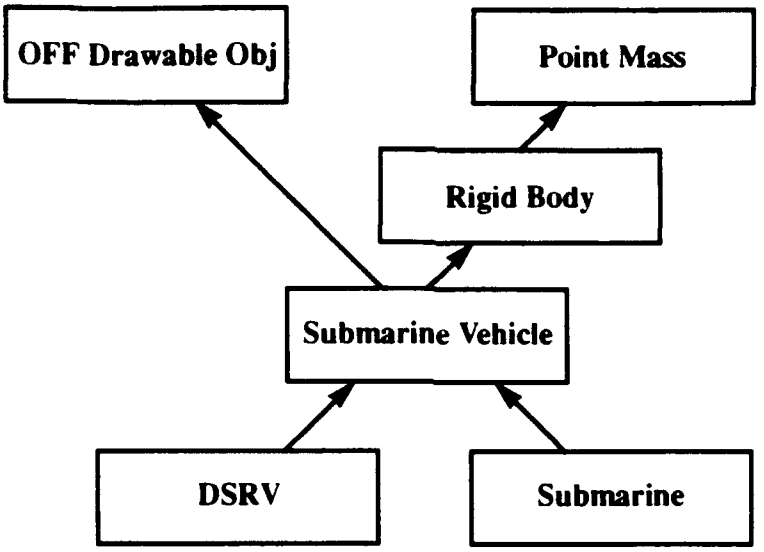

Figure 5 DSRV and Submarine Class Hierarchy

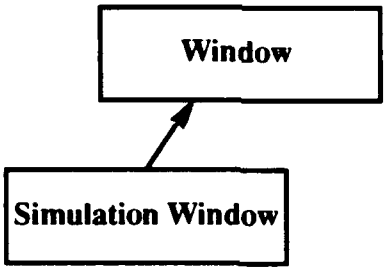

Figure 6 Simulation Window Class Hierarchy

F. OBJECT HIERARCHIES

Object hierarchies reflect a "part of" relationship. In this application there are three object hierarchies of interest: the DSRV, the Simulation Window and the DSRV Simulator.

1. DSRV

The DSRV object uses components which influence its submerged handling characteristics. These components are the propulsor, a ballast system and four thrusters. Other components would be appropriate for defining the capabilities of a DSRV (energy system, sensors, etc.) but they are not directly the focus of this simulation and are therefore not included. Figure 7 shows the DSRV object hierarchy.

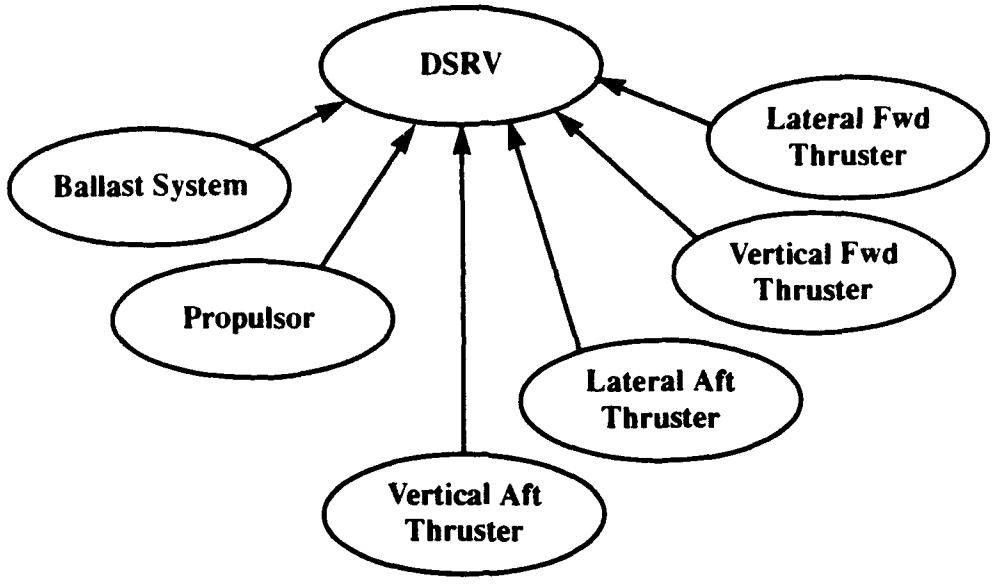

Figure 7 DSRV Object Hierarchy

2. Simulation Window

The Simulation Window is composed of objects which participate in a "part of" hierarchy. They are the requisite views for display and the non-moveable objects which make up the virtual environment. The relationship is reflected in Figure 8.

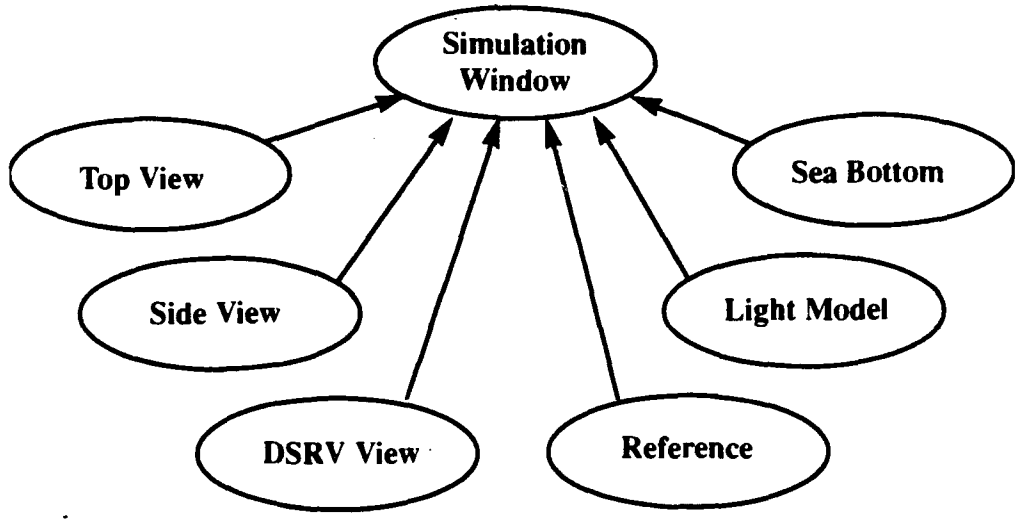

Figure 8 Simulation Window Object Hierarchy

3. DSRV Simulator

The DSRV Simulation is the highest level object in the design. It is composed of objects which allow the DSRV Simulation to operate. These objects include the DSRV object, the Submarine object and the Simulation Window object. Figure 9 shows the relationship.

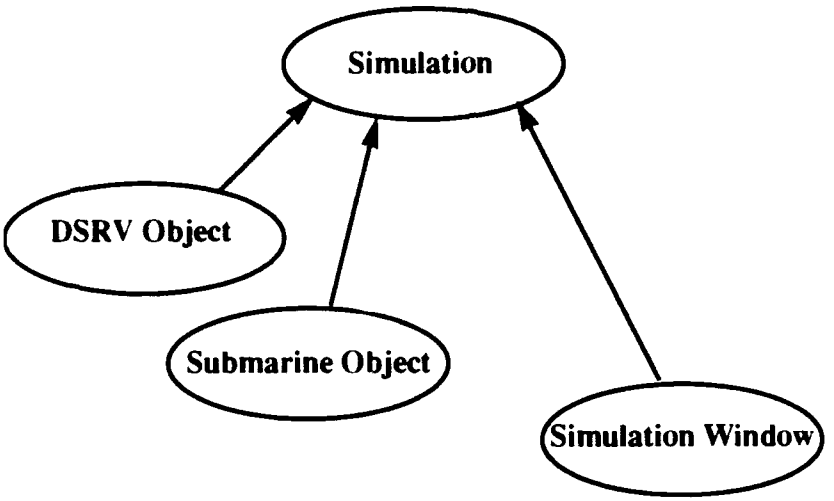

Figure 9 Simulation Object Hierarchy

As expected, the Simulation object hides virtually all of the implementation details. This is in keeping with the guidelines of information hiding and encapsulation which are fundamental to the object oriented analysis and design principle.

G. SUMMARY

The DSRV simulation is the problem domain for the analysis and design phase of the software development. The key abstractions from the problem domain are considered and placed in context with each other through an analysis of their hierarchical class and object hierarchies. The next task is to implement the analysis and design in an appropriate programming language. This is the subject of the next chapter.

V. C++ OBJECT IMPLEMENTATION

A. AT THE TOP, THE MAIN() PROGRAM

The main() program is extremely terse, in keeping with the object oriented discipline. The DSRV_Simulation is the highest level of abstraction and is represented as its own class. The program, main() simply instantiates the DSRV_Simulation and tells it to run, as seen in Figure 10..

```
#include "DSRV_Simulator.H"

void main()
{
   DSRV_Simulation D;
   D.RunSimulation();
}
```

Figure 10 Program main()

B. HIGHEST LEVEL CLASS - DSRV_SIMULATION

This class is also straightforward. It reveals a little more of the design decisions as would be expected at a slightly lower level of the simulation abstraction. In Figure 11, the DSRV_Simulation creates three objects. The DSRV_obj, avalon, is the principle object of the simulation and, as will be seen, is the most complex. The Submarine_obj, thresher, is the target submarine for the simulation. It has limited maneuverability in the simulation. In truth, the submarine will not move at all as it is the target of the DSRV and is assumed to be in distress. The Simulation_Window object, sim_window, is responsible for lighting, viewpoint and the display of static and maneuverable objects.

Notice that the avalon and thresher are instantiated as member variables of a class

```
#include "Dynamic_Objs.H"
#include "Simulation_Window.H"
#include "Sub_Vehicle_Objs.H"

class DSRV_Simulation {

    public:
      DSRV_Simulation();
      ~DSRV_Simulation();

      void RunSimulation();

    private:

      // simulated objects
      DSRV_obj avalon;
      Submarine_obj thresher;

       // window obj
      SimulationWindow sim_window;
};
```

Figure 11 Class DSRV_Simulation

DSRV_Simulation but their graphical display is controlled by the object, sim_window. Sim_window must be provided with the identity of these two moveable objects because sim_window does not instantiate them. This was a design decision to separate the mathematical nature of avalon and thresher from the graphical nature. The two aspects, graphical and mathematical still reside within the same object, but sim_window may call the Draw() member function after sim_window is provided with the identity of the vehicles through pointer references.

C. THE DSRV_OBJ CLASS

The previous chapter discussed the class and object hierarchies for the DSRV object. The DSRV participates in both forms of hierarchy. The class hierarchy starts with the abstraction of a point mass. Point mass uses one of two multi-purpose structures, *ThreeVector* and *ThreeMatrix* (self-explanatory). Rigid body derives from point mass. In turn, SubmergedVehicle derives from rigid body. Each successive class inherits from the

super class using the inheritance mechanism of C++. Class rigid body shows this inheritance mechanism well in Figure 12.

```
class RigidBody: public PointMass, public Torque {

  public:
    RigidBody();
    ~RigidBody();

    void set_inertia (ThreeVector, ThreeVector,
                      ThreeVector);
    void set_orient (double, double, double);
    void set_ang_vel (double, double, double);
     void set_ang_accel (double, double, double);

    ThreeMatrix get_inertia ();
    ThreeVector get_orient ();
    ThreeVector get_ang_vel ();
    ThreeVector get_ang_accel ();

    virtual void fwd_kinematics (long);
     virtual void fwd_dynamics (long);
     virtual void reverse_kinematics ();
     virtual void reverse_dynamics ();
     virtual void update();


  protected:
    ThreeMatrix inertia;
    ThreeVectororient;
    ThreeVectorang_vel;
    ThreeVector ang_accel;
};
```

Figure 12 Class Rigid Body

PointMass and Torque are both inherited by RigidBody. These class definitions can be found in Appendix E. Inheritance allows the sub-class to inherit member data objects and member functions. This is code reuse, an efficiency the C++ language encourages. Combined with virtual functions and dynamic binding, the language encourages polymorphism, another powerful tool for the programmer.

Note the virtual functions for kinematics and dynamics. They allow sub-classes to redefine the functionality, if necessary. This is an example of polymorphism in action.

SubmergedVehicle specializes the rigid body concept to the submerged environment. It introduces the concepts weight, buoyancy and ocean current. These concepts have particular meanings with respect to Newtonian physics. Information hiding is a key ingredient to the object model and it is in this class that the implementation details of these physics concepts are encapsulated.

The Submerged Vehicle class has other interesting properties that are useful, as seen in Figure 13. It diverges from inheriting only physics concepts, to inheriting the class

```
class SubmarineVehicle : public RigidBody,
 public OFF_Drawable_Obj {
  public:
     SubmarineVehicle();
     SubmarineVehicle(char *);
     ~SubmarineVehicle();

     void add_weight (double);
     void add_buoyancy (double);

     void set_weight (double);
     void set_buoyancy (double);
     void set_sea_current (double, double,
double);

     double get_weight ();
     double get_buoyancy ();

     void ready_OFF_file() {OFF_Drawable_Obj::
                            ready_OFF_Obj();};

  protected:

     double weight;
     double buoyancy;
     ThreeVector sea_current;
};
```

Figure 13 Class SubmarineVehicle

properties of an OFF_Drawable_Obj. The Object File Format (OFF) is a useful, object oriented set of utilities for quickly designing, modifying and displaying complex graphical objects. This level of inheritance demonstrates multiple inheritance where the

specialization class assumes properties from indirectly related super classes. This is a powerful tool for developing robust software application solutions.

As an aside, it should be mentioned that network super classes also tie in at this level. They are not shown in Figure 13 but will be brought out in the next chapter when networking the simulator is discussed.

The class DSRV_obj and Submarine_obj both inherit from class Submerged Vehicle. The DSRV_obj is the more interesting of the two. Figure 14 shows the publicly visible declaration of the class. SubmarineVehicle is a super class, and DSRV_obj inherits its

```
class DSRV_obj : public SubmarineVehicle {
  friend class DSRV_Simulation;

  public:

    DSRV_obj();
    DSRV_obj(char *);
    ~DSRV_obj();

    void set_rudder_angle (double);
    void set_sternplane_angle (double);
    void increment_propulsor_rpm (double);
    void stop_propulsor ();

    void toggle_fwd_transverse_thruster (int);
    void toggle_aft_transverse_thruster (int);
    void toggle_fwd_vertical_thruster (int);
    void toggle_aft_vertical_thruster (int);

     void update();
    void Draw();
    void set_image(OBJECT*);
    OBJECT* get_image();

    int ballast_pump_is_on();
    int sea_valve_is_open();
```

Figure 14 Class DSRV_obj, Public Members

abstraction, including the physics, graphics and network capabilities. DSRV_Simulation is a class which works closely with SubmarineVehicle. For efficiency, it has been afforded direct access to SubmarineVehicle members through the C++ *friend* relationship. The

friend construct is not directly a hierarchical issue in terms of "kind of" (class) and "part of" (object) hierarchies.

From the listed member functions, it is clear that DSRV_obj is composed of sub-objects. Figure 15 shows the private members which define the object. The listed member

```
private:

  Ballast_System ballast;
 Thruster *fwd_vertical_thruster;
 Thruster *fwd_transverse_thruster;
 Thruster *aft_vertical_thruster;
 Thruster *aft_transverse_thruster;
 Propulsor *propulsor;

 double dr, ds;)
 double u, v, w;
 double p, q, r;
 double xpos, ypos, zpos;
 double phi, theta, psi;
 double xx[13];
 double M[7][7], Mi[7][7];

 void invert_matrix (double *, double *, int);
 double trapezoid_integration (int,
                               double[], double[]);
};
```

Figure 15 Class DSRV_obj, Private Members

functions provide the interface to the component objects which make up the abstraction of the DSRV. Notice in Figure 15 that the DSRV_obj encapsulates its own abstraction plus the abstractions Ballast_System, Thruster and Propulsor. By declaring objects of these classes as member objects of DSRV_obj, the "part of" relationship is fulfilled.

D. THE SIMULATION_WINDOW CLASS

The Simulation _Window class encapsulates the abstraction from the analysis and design activity dealing with the chores of producing the graphical representation of the simulated objects. As noted earlier, the DSRV object and Submarine object are created by the DSRV_Simulation class object which is not concerned with the details of how the objects are displayed, only with the timing. Accordingly, the Simulation_Window is given

the identity of the simulated objects in the form of pointers which then may be used to invoke the Draw() methods of the individual objects. As seen in Figure 16, the Simulation_Window object has a public member function which allows the identity of the simulated objects to be made known to the Simulation_Window called *PassDrawObj()*.

```
class SimulationWindow : public Window {

  public:
    SimulationWindow ();
    ~SimulationWindow();

    void PassDrawObj(DSRV_obj *, Submarine_obj *);

    // define virtual functions from class Window
    void evaluate_menu_selection();
    void DrawWindow ();

  private:
    viewSelect viewselection;
    boolean selectgrid;

    DSRV_obj *dsrv;
    Submarine_obj *sub;
    OBJECT *light_obj; // OFF objects
    OBJECT *ref_obj;
    OBJECT *floor;

    static float fog[5];

    // define virtual functions from class Window
    long makethemenus();
    void processmenuhit();
};
```

Figure 16 Class Simulation_Window

This member function takes address arguments which are assigned to internal member pointers.

Also note that SimulationWindow redefines inherited member functions, *evaluate_menu_selection()*, *DrawWindow()*, *makethemenus()*, and *processmenuhit()*. This is possible through polymorphism afforded by the virtual function definitions in the generalized super class Window.

The full class definitions and implementations are found in the appendices.

E. MATH MODEL APPLICATION

The C++ code which implements the mathematical model was tailored from a simulation of the Autonomous Underwater Vehicle (AUV) written in C [Marco91]. The state variable structure and general algorithm were very helpful. The AUV control, propulsion and handling coefficients are markedly different from those of the DSRV and therefore were not transferrable.

F. SUMMARY

The C++ object implementation is created from the design decisions made earlier in the analysis and design process. The implementation details adhere to the design by using the object oriented facilities of the C++ programming language. In this manner, the development process creates an application which is true to the object paradigm and easily extensible. The flexibility of this methodology is demonstrated by the ease with which the classes for the network interface are tailored to the DSRV simulator in the next chapter.

VI. NETWORK ISSUES

A. NETWORK STUB

Although the DSRV simulation is not networked in its current form, the whole concept of the simulation depends upon an eventual tie-in to a networked environment. Indeed, the network aspect of the simulator is part of the total analysis and design activity. Therefore, the network stubs are included in the simulator architecture through an object hierarchy relationship in the SubmergedVehicle class. Recall that this is where the graphics and physics abstractions were also brought together.

The network architecture follows the model developed for DARPA by the Department of Computer Science at the University of Central Florida [Blau92]. This architecture is an object oriented networked environment called the Virtual Environment Real-time Network (VERN). VERN is written in the C++ language using the object model, which makes it ideal for this simulation application.

VERN establishes a class hierarchy as seen in Figure 17. The leaf nodes are abstract classes which must be made object classes through inheritance. In particular, concrete sub-classes must be declared for *AbstractPlayer*, *AbstractGhost*, and *AbstractState*.

The DSRV simulator declares two such classes, Network_Player and Network_Ghost. They are the stubs designed into the architecture for a functional network interface. They must be expanded into a fully integrated structure in a future extension of the simulator. The AbstractState class is not represented with a stub because the full state information is encapsulated in DSRV_obj. This too must be expanded to integrate fully into a network.

Under the VERN network model, all moving objects in the virtual environment have a source node. This node spawns the object and keeps track of its detailed state information. This is called a "Player" object. The Player also maintains a "Ghost" object. The Ghost maintains a dead reckoning solution for the dynamic object. The Player is responsible for

comparing the detailed model with the Ghost model and initiating a message packet to the network if an arbitrary error threshold is exceeded.

Each node on the network also maintains a Ghost for each dynamic object. It is identical to the Player's Ghost with the same dead reckoning functionality. Remote node Ghosts receive corrected position and velocity data from the Player when the Player determines that the appropriate error thresholds have been exceeded.

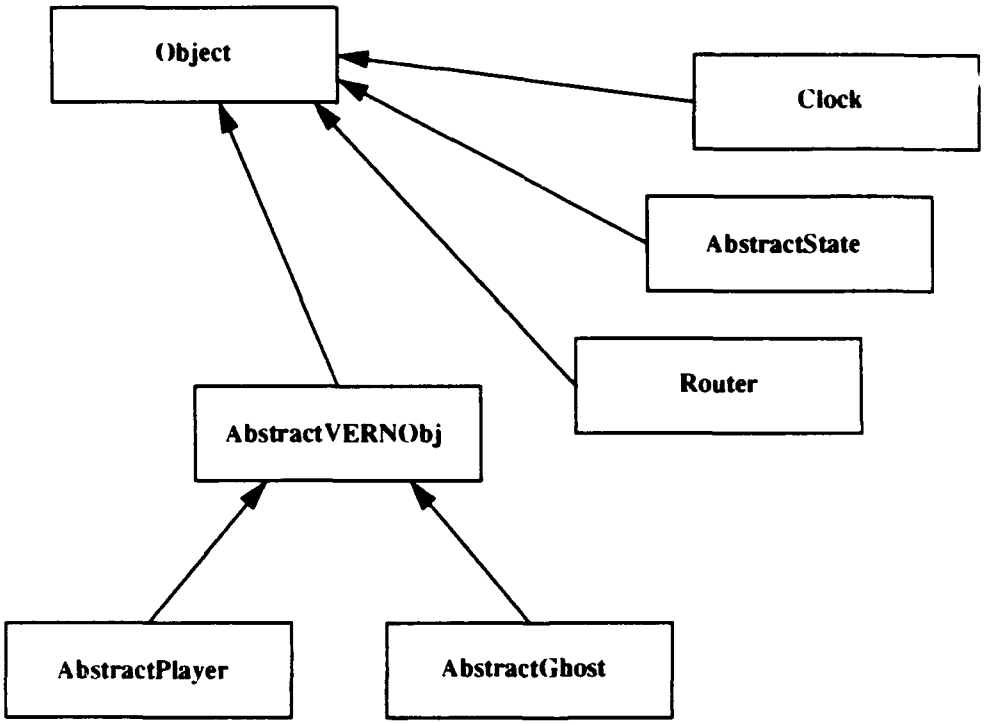

Figure 17 VERN Network Class Hierarchy

The network class stubs with minimal member function definitions are shown in Figure 18.

```
class Network_Player {
  public:
    Network_Player();
    Network_Player(char *n);
    ~Network_Player();

    void set_error_tolerance(double, double, double);
    void processMsg();
    void computeNextState(double, double, double);

  private:
    char *name;
    ThreeVector error_tolerance;
};



class Network_Ghost {
  friend Network_Player;
  public:
    Network_Ghost();
    ~Network_Ghost();

    void send_update();
    void receive_update();
    void computeNextState(float);
    void update_state() {};

  protected:
    ThreeVector net_position;
    ThreeVector net_velocity;
};
```

Figure 18 Network Class Stubs

B. NETWORK ERROR THRESHOLDS

Typically the Player object will use a positional error threshold to determine when it is appropriate to inform all Ghosts to update their mini-state information. Ten percent of body length is a common threshold for issuing an update message packet. This works fine for relatively fast moving objects, such as land vehicles and aircraft. The DSRV moves slowly in relation to such active objects and may move for quite some time before

exceeding the threshold. Indeed, the DSRV operates for long periods in hover mode making small corrections to position and attitude.

Most networks have a time threshold for communications. If a network object issues no messages in a preset interval, the object is considered dormant and may be killed or removed from the address list. The slow moving DSRV may fall prey to this algorithm and so should issue an update message, say every five seconds, for a seven second cutoff window. This would be built into the Network_Player class

C. MULTIPLE CONCURRENT VIEWS

The DSRV simulator can take great advantage of a networked environment. It already has multiple views installed for the operator. These views would be helpful in using the simulator as a training tool. The operator would control the DSRV from the view within the DSRV. An instructor or evaluator would monitor the approach and maneuvering expertise of the operator from an external view.

Further, multiple users could be networked in a training environment dispersed geographically. DSRV operators in Charleston, South Carolina (where the east coast DSRV teams are based) could hold networked training sessions with their west coast counterparts in San Diego, California. This is the type of training leverage which will be increasingly in demand in an environment of tight budgets and dispersed centers of specialized expertise.

D. SUMMARY

The DSRV simulator should be networked to achieve its full utility as a distributed training and simulation environment. The architecture for establishing a network interface was discussed in this chapter. The operator interface and graphical visualization capabilities will be discussed in the next chapter.

VII. INTERFACE AND VISUALIZATION

A. OPERATOR INTERFACE

The DSRV simulator uses the SpaceBall, keyboard and window menus as input devices. The SpaceBall provides the interface for DSRV maneuvering controls. The keyboard provides rudimentary control functions for the submarine, control of the ballast pump on the DSRV, and an escape key to end the simulation. The menus allow the operator to engage a preset ocean current effect, select the viewpoint for display, and toggle a reference grid around the DSRV -- useful for gauging DSRV attitude.

Output variables may be viewed through the console window which starts the simulation.

1. Spaceball

The SpaceBall "Z" transverse input is used for controlling propulsor thrust. The keyboard "S" key stops the propulsor. The SpaceBall "Z" rotation axis controls propulsor rotation about the DSRV body "Y" axis (angle of attack, as simulated by a stern plane control surface). It is the primary cruising attitude control mechanism. The SpaceBall "Y"axis rotation controls propulsor azimuth rotation (as simulated by a rudder control surface).

All eight SpaceBall buttons are used to initiate thruster commands. The four end buttons are used to control the transverse thruster commands. The top two end buttons

control the forward transverse thruster. The bottom two end buttons control the aft transverse thrusters.

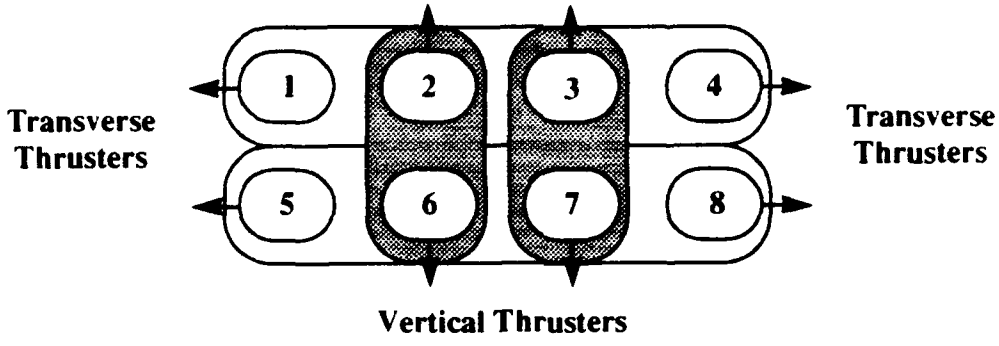

Figure 19 SpaceBall Button Commands

The middle four SpaceBall buttons control the vertical thrusters. The top two middle buttons are for commanding the forward and after vertical thrusters to operate in the negative "Z" body axis (up). The bottom two middle buttons are for commanding the forward and after vertical thrusters to operate in the positive "Z" body axis (down).

2. Keyboard

The "P" key energizes the DSRV ballast pump. Using the up and down arrow increases the ballast pump rate. Pressing the "P" again, stops the ballast pump. The "F" key operates the DSRV ballast system sea valve. Pressing the up arrow key opens the orifice wider, effectively increasing the flood rate. Pressing the down arrow key reduces the sea valve orifice setting and reduces the flood rate into the ballast system.,

The left arrow key and right arrow key command the submarine ordered course to increase or decrease in ten degree increments. Submarine speed commands are single step change commands using the "Q" key to command speed to 4 knots, and the "A" key to order all stop.

3. Menus

The menus system allows the operator to select a preset current. It also allows the operator to disengage or engage the DSRV reference grid. The reference grid provides a visual reference for the initial attitude of the DSRV. Finally, the menus system allows the operator to select the desired viewpoint.

All user interface commands are summarized in Table 4:.

Table 4: USER INTERFACE COMMANDS

Interface	Function
SpaceBall Forward	Increase propulsor RPM
SpaceBall Back	Decrease propulsor RPM
Keyboard "S" Key	Stop propulsor
SpaceBall Rotate Left	Rudder angle causes a turn to the left
SpaceBall Rotate Right	Rudder angle causes a turn to the right
SpaceBall Button 1	Forward transverse thruster pushes left
SpaceBall Button 4	Forward transverse thruster pushes right
SpaceBall Button 5	Aft transverse thruster pushes left
SpaceBall Button 8	Aft transverse thruster pushes right
SpaceBall Button 3	Forward vertical thruster pushes up
SpaceBall Button 7	Forward vertical thruster pushes down
SpaceBall Button 2	Aft vertical thruster pushes up
SpaceBall Button 6	Aft vertical thruster pushes down
Keyboard "P" Key	Start/stop ballast pump
Keyboard "F" Key	Open/shut sea valve
Keyboard Up arrow	Increase pump rate or flood rate
Keyboard Down Arrow	Decrease pump rate or flood rate
Menu "Toggle Current"	Starts/stops a preset ocean current
Menu "Reference Grid	Engages/disengages reference grid
Keyboard "Esc"	Stop the simulation

B. VISUALIZATION

The visualization facilities are spartan. The DSRV is defined in a file using the Object File Format (OFF). It is composed of 108 polygons which were created off line using the OFF object construction tool. The file encapsulates material, color and polygon definitions. The results are shown in grey scale image in Figure 20.

Figure 20 DSRV Graphical Object From OFF File

The light model, submarine and reference object (a water tower for maneuvering reference) are all OFF objects. A reference grid about the DSRV is created by the Sumulation_Window class object using direct SGI draw commands.

C. SUMMARY

The simulator operator interface is very simple. It provides a basic set of controls for adjusting the DSRV in both cruise and hover modes. Controls are provided for the propulsor, the ballast system and the four thrusters.

VIII. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

To some extent, the value of this work will not be known until it is extended by other students. One of the aims has been to use object oriented techniques to develop the DSRV simulator. While the simulator is a success from the standpoint that it faithfully models and displays the handling characteristics of the DSRV, the real power of the object oriented approach is the tacit understanding that software reuse is paramount to the future of developing complex systems and a greater success awaits the extension of this simulation software to other submerged vehicles and other simulators.

B. RECOMMENDATIONS

1. Reuseable "Simulator" Base Class

Following the object oriented design, it would be beneficial if the Simulation class were placed in a class hierarchy where common features of a generic "simulator" could be factored out. This would aid in the construction of other, non-underwater vehicle simulators. Such an analysis and design would make the creation of another simulator as easy as extending (through inheritance) and instantiating a simulation object.

2. Extend Default Class Functionality

The class of basic physics objects have limited functionality. The virtual functions that are defined use rudimentary models and unsophisticated mathematical methods. A more functional set of physics objects would be invaluable in modeling increasingly complex abstractions with relatively little effort.

3. Encapsulate Constraint Relationships In Classes

Constraint relationships allow physical objects to self-monitor themselves to ensure they obey well understood rules of behavior. This set of physics objects makes no pretense of attempting to constrain behavior by following such phenomenon as energy conservation and constraint conditions. Such an extension to the physics objects would make their use in modeling real systems much more robust.

4. Interface And Display

The user interface is not well developed. While much effort was expended to faithfully model the handling characteristics of the DSRV, the user interface is bare bones and the operator control and display system non-existent. Such an extension to the simulator could be faithfully constructed using the object oriented analysis and design techniques used in this simulator.

5. Networked Virtual World

To repeat, the network interface of this simulator is rudimentary but necessary to show how the interface blends in with the overall simulator architecture. The eventual networking of the DSRV is where the real practical benefits of the simulator can come to fruition. It is the eventual goal of an application such as this to be useful in a distributed training and analysis environment afforded by a networked system.

6. Better Numerical Integration Method

The mathematical model would be well served by the use of a more accurate numerical integration method. This simulation uses the trapezoid integration method. It might be appropriate to introduce the fourth order Runge-Kutta numerical method [McGhee75], for example, although further simulation studies will be required to determine if this is indeed the case.

7. Collision Detection

Perhaps the weakest part of the simulation is the lack of a collision detection method. While the DSRV handling characteristics are accurate, the simulation loses its "reality" sense when objects may go through one another without stopping. The object oriented paradigm makes such an extension to the model well bounded and with little likelihood of introducing unintended side effects in other parts of the simulator architecture.

C. FINAL REMARKS

Computer simulation is a valuable tool for extending human understanding and leveraging our resources. The complexity of the problems that are addressed with computers is growing. Software complexity grows accordingly. Therefore, the analysis, design and implementations must be closely aligned to get maximum use of our software tools. This thesis takes the simulation problem domain and applies the tools made possible by the object oriented design method and the object oriented capabilities of the C++ language. There are many fruitful areas of research and extension which may spring from this study.

LIST OF REFERENCES

[Badler91] Badler, N. I., Barsky, B. A., and Zeltzer, D., eds., *Making Them Move: Mechanics, Control, and Animation of Articulated Figures*, Morgan Kaufmann Publishers, Inc., 1991.

[Barzel92] Barzel, R., *Physically Based Modeling for Computer Graphics. A Structured Approach*, Academic Press, Inc., 1992.

[Blau92] Blau, B., C. E. Hughes, J. M. Moshell, J. Michael and C. Lisle, "Networked Virtual Environments", *1992 Symposium on Interactive 3D Graphics*, 30 March, 1992, pp. 157-164.

[Booch91] Booch, G., *Object Oriented Design with Applications*, The Benjamin/ Cummings Publishing Company, Inc., 1991.

[Cantu92] Cantu, M. and S. Tendon, *Borland C++ 3.1 Object - Oriented Programming*, Bantam Books, 1992.

[Comstock67] Comstock, J. P., ed., *Principles of Naval Architecture*, The Society of Naval Architests and Marine Engineers, New York, 1967.

[Cooke92] Cooke, J. M., *NPSNET: Flight Simulation Dynamic Modeling Using Quaternions*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1992.

[Finney90] Finney, R. L. and G. B. Thomas, Jr., *CALCULUS*, Addison - Wesley Publishing Company, 1990.

[Foley87] Foley, J. D., A. van Dam, S. K. Feiner, J. F. Hughes, *Computer Graphics Principles and Practice*, Addison-Wesley Publishing Company, 1987.

[Healey92] Healey, A. J., *Dynamics of Marine Vehicles*, course notes for the Naval Postgraduate School course ME 4823, "Dynamics of Marine Vehicles," 1992.

[Marco91] Marco, D., "C" source code for computer simulation of the Autonomous Underwater Vehicle (AUV), Department of Mechanical Engineering, Naval Postgraduate School, Monterey, CA, 1991.

[McGhee75] McGhee, R. B., *Why Be Afraid of Numerical Integration?*, Course notes dated January 17, 1975.

[McGhee91] McGhee, R. B., *A Simplified Dynamic Model for Horizontal Plane Maneuvering by the NPS Model 2 AUV*, Teaching notes dated March 19, 1991.

[NSRDC67] Naval Ship Research and Development Center Report 099-H-05, , *Performance of a DSRV Propeller in Four Modes of Vehicle Operation (NSRDC Model 5128)* by Beveridge, J. L., and F. W. Puryear, August 1967.

[NSRDC69] Naval Ship Research and Development Center Report 3030, *Model Investigation of the Stability and Control Characteristics of the Contract Design for the Deep Submergence Rescue Vehicle (DSRV)* by D. B. Young, April 1969.

[Rheingold91] Rheingold, H., *Virtual Reality*, Simon & Schuster, 1991.

[Weidner75] Weidner, R.T. and R. S. Sells, *Elementary Physics: Classical and Modern*, Allyn and Bacon, Inc., 1975.

[Yourdon89] Yourdon, E., *Modern Structured Analysis*, Prentice Hall, Inc., 1989

APPENDIX A

MAIN() PROGRAM

```
#include "DSRV_Simulator.H"

void main()
{
  DSRV_Simulation D;
  D.RunSimulation();
}
```

APPENDIX B

DSRV SIMULATOR

```
// files for this application
#include "Dynamic_Objs.H"
#include "Simulation_Window.H"
#include "Sub_Vehicle_Objs.H"

//////////////////////////////////////////////////////////////////
//
//  DSRV_Simulation.H
//
//////////////////////////////////////////////////////////////////

class DSRV_Simulation {

  public:
    DSRV_Simulation();
    ~DSRV_Simulation();

    void RunSimulation();

  private:

    // simulated objects
    DSRV_obj          avolon;
    Submarine_obj     thresher;

    // window obj
    SimulationWindow  sim_window;
};

//////////////////////////////////////////////////////////////////
//
//  DSRV_Simulation.c
//
//////////////////////////////////////////////////////////////////
```

```
#include <stdio.h>
#include <gl.h>
#include <device.h>

// files for calls to time clock
#include <sys/types.h>
#include <sys/times.h>
#include <sys/param.h>

// files for OFF function calls
#include "image_types.H"
#include "rdobj_opcodes.H"

// files for this application
#include "DSRV_Simulator.H"
//#include "Simulation_Variables.H"

#define LEFT  -1
#define RIGHT  1
#define UP    -1
#define DOWN   1


//////////////////////////////////////////////////////////////////
///////
//
//  DSRV_Simulation
//
//////////////////////////////////////////////////////////////////
///////

DSRV_Simulation:: DSRV_Simulation() {

  // queue the redraw device
  qdevice(REDRAW);

  // queue the menubutton (right button)
  qdevice(MENUBUTTON);

  // use keyboard input signals
  qdevice(LEFTARROWKEY);
  qdevice(RIGHTARROWKEY);
  qdevice(UPARROWKEY);
```

```
  qdevice(DOWNARROWKEY);
  qdevice(ESCKEY);
  qdevice(PKEY);
  qdevice(FKEY);
  qdevice(SKEY);

  // use spaceball device buttons
  qdevice(SBBUT1);
  qdevice(SBBUT2);
  qdevice(SBBUT3);
  qdevice(SBBUT4);
  qdevice(SBBUT5);
  qdevice(SBBUT6);
  qdevice(SBBUT7);
  qdevice(SBBUT8);

  // use spaceball control signals
  qdevice(SBTZ);
  qdevice(SBTY);
  qdevice(SBRY);
  qdevice(SBRZ);

  // adjust sensitivity for devices
  noise(SBTZ, 10);
  noise(SBTY, 10);
  noise(SBRY, 10);
  noise(SBRZ, 10);

};

DSRV_Simulation:: ~DSRV_Simulation() {};

void DSRV_Simulation:: RunSimulation() {

  avolon.ready_OFF_file();
  thresher.ready_OFF_file();

  // send identity of simulated objects to the sim window
sim_window.PassDrawObj(&avolon, &thresher);


  // boolean for primary loop
  int running = 1;
```

```
// boolean for operating ballast system
int pumping  = 0;
int flooding = 0;

// value returned from the event queue
short event_val;

// DSRV rudder angle value
double rudder_angle = 0.0;

while(running) {

  // check event queue for operator commands
  while(qtest()) {

    switch(qread(&event_val)) {

      case MENUBUTTON:

        if(event_val == 1) {

          // send message to window to check menu selection
          sim_window.evaluate_menu_selection();

        }
        break;

      case RIGHTARROWKEY:

        if(event_val == 1) {
          sub1.velocity[X] += 0.1;
          sub1.velocity[Y] += 0.1;
        }
        break;

      case LEFTARROWKEY:

        if(event_val == 1) {
          sub1.velocity[X] -= 0.1;
          sub1.velocity[Y] -= 0.1;
        }
        break;
```

```
case UPARROWKEY:

 // increase ballast pump rate
 if(event_val == 1) {
 avolon.ballast.change_pump_rate((double)event_val);
  }
  break;

case DOWNARROWKEY:

  // decrease ballast pump rate
  if(event_val == 1) {
    avolon.ballast.change_pump_rate
                  (-(double)event_val);
  }
  break;

case ESCKEY:

  running = 0;
  break;

case PKEY:

  if (pumping) {
    avolon.ballast.stop_pump();
    pumping = 0;
  }
  else {
    avolon.ballast.start_pump();
    pumping = 1;
  }
  break;

case FKEY:

  if (flooding) {
    avolon.ballast.close_sea_valve();
    flooding = 0;
  }
  else {
    avolon.ballast.open_sea_valve();
```

```
    flooding = 1;
  }
  break;

case SKEY:

  if (event_val == 1) avolon.stop_propulsor();
  break;

 case SBBUT1:

  if (event_val == 1) {
    avolon.toggle_fwd_transverse_thruster(LEFT);
  }
  break;

case SBBUT2:

  if (event_val == 1) {
    avolon.toggle_fwd_vertical_thruster(UP);
  }
  break;

case SBBUT3:

  if (event_val == 1) {
    avolon.toggle_aft_vertical_thruster(UP);
  }
  break;

case SBBUT4:

  if (event_val == 1) {
    avolon.toggle_fwd_transverse_thruster(RIGHT);
  }
  break;

case SBBUT5:

  if (event_val == 1) {
    avolon.toggle_aft_transverse_thruster(LEFT);
  }
  break;
```

```
case SBBUT6:

  if (event_val == 1) {
    avolon.toggle_fwd_vertical_thruster(DOWN);
  }
  break;

case SBBUT7:

  if (event_val == 1) {
    avolon.toggle_aft_vertical_thruster(DOWN);
  }
  break;

case SBBUT8:

  if (event_val == 1) {
    avolon.toggle_aft_transverse_thruster(RIGHT);
  }
  break;

case SBTZ:

  avolon.increment_propulsor_rpm(double(event_val)
                                    / 10000.0);
  break;

case SBTY:

 break;

case SBRY:

  avolon.set_rudder_angle(event_val / 10000.0);
  break;

case SBRZ:

  avolon.set_sternplane_angle(-event_val / 10000.0);
  break;

case REDRAW:
```

```
            reshapeviewport();
            break;

          default:
           break;

        }    //end switch on event queue item
      }    // endif qtest()

      // update the internal state of the simulated objects
      avolon.update();
      thresher.update();

      // draw the environment
      sim_window.DrawWindow();

    }    // end while()
};   // end RunSimulation()
```

APPENDIX C

SUB VEHICLE OBJECTS

```
#include "Dynamic_Objs.H"
#include "Window.H"
#include "Network_Interface.H"

#ifndef _SUB_VEHICLE_OBJS
#define _SUB_VEHICLE_OBJS


//////////////////////////////////////////////////////////////////////
//////
//
//  Ballast_System
//
//////////////////////////////////////////////////////////////////////
////,

class Ballast_System {

  public:
    Ballast_System ();
    Ballast_System(double, double, double, double);
    ~Ballast_System();

    void start_pump ();
    void stop_pump  ();
    void change_pump_rate (double);
    void open_sea_valve ();
    void close_sea_valve ();
    void adjust_sea_valve_orifice (double);
    void update_ballast(float);

    void set_max_ballast     (double);
    void set_max_pump_rate   (double);
    void set_max_flood_rate  (double);
    void set_current_ballast (double);
```

```
    int pump_is_on();
    int sea_valve_is_open ();
    double get_current_ballast ();

  private:
    int    pump_on;
    int    sea_valve_open;
    double max_ballast;
    double max_pump_rate;
    double max_flood_rate;
    double current_ballast;
    double current_pump_rate;
    double current_flood_rate;
};


//////////////////////////////////////////////////////////////////
//////////
//
//  Thruster
//
//////////////////////////////////////////////////////////////////
//////////

class Thruster {

  public:
    Thruster();

    // max magnitude, location x, y, z
    Thruster (double, double, double, double);
    ~Thruster();

    virtual void start_positive_thrust ();
    virtual void start_negative_thrust ();
    virtual void stop_thruster ();
    virtual void change_thruster_rate (double);
    virtual void update_thruster_force(float);

    virtual void set_max_thrust         (double);
    virtual void set_thruster_location  (double, double,
double);
```

```
    virtual void set_thruster_direction (double, double,
double);
    virtual void set_screw_efficiency   (double);
    virtual void set_thruster_rpm       (double);

    virtual ThreeVector get_thrust ();
    virtual ThreeVector get_torque ();


  protected:
    int          thruster_on;
    double       rpm;
    double       screw_efficiency;  //values 0.0..1.0
    double       max_thrust;
    ThreeVector body_location;
    Force        thruster_force;
    Torque       thruster_torque;
    ThreeVector absolute_force_vector;
    ThreeVector absolute_torque_vector;

};

//////////////////////////////////////////////////////////////
//////////
//
//  Propulsor
//
//////////////////////////////////////////////////////////////
//////////

class Propulsor : public Thruster {

  public:
    Propulsor();
    ~Propulsor();

    void increment_propulsor_rpm (double);
    void set_propulsor_rpm       (double);
    void update_thruster_force   (float);

};
```

```
//////////////////////////////////////////////////////////////
//
//  SubmarineVehicle
//
//////////////////////////////////////////////////////////////

class SubmarineVehicle : public RigidBody,
                         public OFF_Drawable_Obj {
  public:
    SubmarineVehicle();
    SubmarineVehicle(char *);
    ~SubmarineVehicle();

    void add_weight   (double);
    void add_buoyancy (double);

    void set_weight   (double);
    void set_buoyancy (double);
    void set_sea_current (double, double, double);

    double get_weight   ();
    double get_buoyancy ();

    void ready_OFF_file() {OFF_Drawable_Obj::
ready_OFF_Obj();};

  protected:

    double      weight;
    double      buoyancy;
    ThreeVector sea_current;

    // network interface stubbs for development
//    Network_Player network_player;
//    Network_Ghost  network_ghost;

};

//////////////////////////////////////////////////////////////
//
```

```
//  DSRV_obj
//
////////////////////////////////////////////////////////////////////////
////////

class DSRV_obj : public SubmarineVehicle {

  friend class DSRV_Simulation;

  public:
    DSRV_obj();
    DSRV_obj(char *);
    ~DSRV_obj();

    void set_rudder_angle     (double);
    void set_sternplane_angle (double);
    void increment_propulsor_rpm (double);
    void stop_propulsor ();


    void toggle_fwd_transverse_thruster (int);
    void toggle_aft_transverse_thruster (int);
    void toggle_fwd_vertical_thruster   (int);
    void toggle_aft_vertical_thruster   (int);

    void update();
    void Draw();
    void set_image(OBJECT*);
    OBJECT* get_image();

    int ballast_pump_is_on();
    int sea_valve_is_open();

  private:

    Ballast_System ballast;
    Thruster       *fwd_vertical_thruster;
    Thruster       *fwd_transverse_thruster;
    Thruster       *aft_vertical_thruster;
    Thruster       *aft_transverse_thruster;
    Propulsor      *propulsor;

    OBJECT         *propeller;
```

```
    double dr, ds;      // rudder and stern planes (for plane
sim)
    double u, v, w;
    double p, q, r;
    double xpos, ypos, zpos;
    double phi, theta, psi;
    double xx[13];
    double M[7][7], Mi[7][7];

    void   invert_matrix (double *, double *, int);
    double trapezoid_integration(int, double[], double[]);

};

////////////////////////////////////////////////////////////
//

//   Submarine_obj
//
////////////////////////////////////////////////////////////
//

class Submarine_obj : public SubmarineVehicle {

  public:
    Submarine_obj();
    Submarine_obj(char*);
    ~Submarine_obj();

    void update();
    void Draw();
    void set_image(OBJECT*);
    OBJECT* get_image();

  private:
    OBJECT *propeller;
    double prop_rot;

};

#endif
```

```
//////////////////////////////////////////////////////////////////////
//
//    Sub_Vehicle_Objs.C
//
//////////////////////////////////////////////////////////////////////


#include <stdio.h>
#include <math.h>

//#include <iostream.h>

#include "Simulation_Variables.H"
#include "Sub_Vehicle_Objs.H"
#include "image_types.H"
#include "rdobj_opcodes.H"


//////////////////////////////////////////////////////////////////////
//////
//
//    Ballast_System
//
//////////////////////////////////////////////////////////////////////
////

Ballast_System:: Ballast_System () {};

Ballast_System:: Ballast_System (double max_ball,  double
curr_ball,
                                 double max_pump, double
max_flood) {
  max_ballast        = max_ball;
  max_pump_rate      = max_pump;
  max_flood_rate     = max_flood;
  current_ballast    = curr_ball;
  current_pump_rate  = 10.0;        // initially 10 lbs per
minute
  current_flood_rate = 10.0;        // initially 10 lbs per
minute
  pump_on            = 0;           // initially pump is off
```

```
  sea_valve_open      = 0;          // initially sea_valve is
closed
};

Ballast_System:: ~Ballast_System() {};

void Ballast_System:: start_pump() {
  pump_on = 1;
};

void Ballast_System:: stop_pump() {
  pump_on = 0;
};

void Ballast_System:: change_pump_rate(double rate) {
  current_pump_rate += rate;
  if (current_pump_rate > max_pump_rate) current_pump_rate =
max_pump_rate;
};

void Ballast_System:: open_sea_valve () {
  sea_valve_open = 1;
};

void Ballast_System:: close_sea_valve () {
  sea_valve_open = 0;
};

void Ballast_System:: adjust_sea_valve_orifice (double rate)
{
  current_flood_rate += rate;
  if (current_flood_rate > max_flood_rate)
current_flood_rate = max_flood_rate;
};

void Ballast_System:: update_ballast(float dt) {
  if (pump_on) current_ballast -= current_pump_rate * dt
60.0;
  if (sea_valve_open) current_ballast += current_flood_rate
* dt / 60.0;
};

void Ballast_System:: set_max_ballast (double max) {
```

```
  max_ballast = max;
};

void Ballast_System:: set_max_pump_rate (double max) {
  max_pump_rate = max;
};

void Ballast_System:: set_max_flood_rate (double max) {
  max_flood_rate = max;
};

void Ballast_System:: set_current_ballast (double ball) {
  if (ball > max_ballast) current_ballast = max_ballast;
  if (ball < 0.0) current_ballast = 0.0;
  else current_ballast = ball;
};

int Ballast_System:: pump_is_on() {
  return pump_on;
};

int Ballast_System:: sea_valve_is_open() {
  return sea_valve_open;
};

double Ballast_System:: get_current_ballast() {
  return current_ballast;
};

////////////////////////////////////////////////////////////////
//
//  Thruster
//
////////////////////////////////////////////////////////////////

Thruster:: Thruster()  {
  thruster_on = 0;
  screw_efficiency = 1.0;
};
```

```
Thruster:: Thruster (double max, double x, double y, double
z) {
  max_thrust        = max;
  screw_efficiency = 1.0;
  body_location.x   = x;
  body_location.y   = y;
  body_location.z   = z;
};

Thruster:: ~Thruster() {};

void Thruster:: start_positive_thrust () {
  if (thruster_on == 1) thruster_on = 0;
  else thruster_on = 1;
};

void Thruster:: start_negative_thrust () {
  if (thruster_on == -1) thruster_on =  0;
  else thruster_on = -1;
};

void Thruster:: stop_thruster () {
  thruster_on = 0;
};

void Thruster:: change_thruster_rate(double r) {
  rpm += r;
};

void Thruster:: update_thruster_force(float dt) {

  thruster_force.set_magnitude(screw_efficiency * rpm *
fabs(rpm) *dt / 60.0);

  if (thruster_force.get_magnitude() >  max_thrust) {
    thruster_force.set_magnitude (max_thrust);
  }

  if (thruster_force.get_magnitude() <  -max_thrust) {
    thruster_force.set_magnitude (-max_thrust);
  }
```

```
  // thruster_on may take values -1, 0, 1 to denote thruster
direction
  //   and operation status
  absolute_force_vector.x = thruster_force.get_magnitude()
                            *
thruster_force.get_components().x
                            * double(thruster_on);
  absolute_force_vector.y = thruster_force.get_magnitude()
                            *
thruster_force.get_components().y
                            * double(thruster_on);

  absolute_force_vector.z = thruster_force.get_magnitude()
                            *
thruster_force.get_components().z
                            * double(thruster_on);

  absolute_torque_vector.x = absolute_force_vector.y *
body_location.z +
                             absolute_force_vector.z *
body_location.y;
  absolute_torque_vector.y = absolute_force_vector.z *
body_location.x +
                             absolute_force_vector.x *
body_location.z;
  absolute_torque_vector.z = absolute_force_vector.y *
body_location.x +
                             absolute_force_vector.x *
body_location.y;
};

void Thruster:: set_max_thrust (double max) {
  max_thrust = max;
};

void Thruster:: set_thruster_location (double x, double y,
double z) {
  body_location.x = x;
  body_location.y = y;
  body_location.z = z;
};
```

```
void Thruster:: set_thruster_direction (double i, double j,
double k) {
  thruster_force.set_components (i, j, k);
};

void Thruster:: set_screw_efficiency (double cs) {
  screw_efficiency = cs;
  if (screw_efficiency > 1.0) screw_efficiency = 1.0;
  if (screw_efficiency < 0.0) screw_efficiency = 0.0;
};

void Thruster:: set_thruster_rpm(double r) {
  rpm = r;
};

ThreeVector Thruster:: get_thrust() {
  return absolute_force_vector;
};

ThreeVector Thruster:: get_torque() {
  return absolute_torque_vector;
};

////////////////////////////////////////////////////////////////
//////
//
// Propulsor
//
////////////////////////////////////////////////////////////////
//////

Propulsor:: Propulsor() {};

Propulsor:: ~Propulsor() {};

void Propulsor:: increment_propulsor_rpm (double r) {
  rpm += r;
};

void Propulsor:: set_propulsor_rpm (double r) {
  rpm = r;
};
```

```
void Propulsor:: update_thruster_force(float dt) {

  thruster_force.set_magnitude(screw_efficiency * rpm *
fabs(rpm) * dt / 60.0);

//(xprop/2.0)*(n_rpm*fabs(n_rpm))

  if (thruster_force.get_magnitude() >  max_thrust) {
    thruster_force.set_magnitude (max_thrust);
  }

  if (thruster_force.get_magnitude() <  -max_thrust) {
    thruster_force.set_magnitude (-max_thrust);
  }

  // thruster_on may take values -1, 0, 1 to denote thruster
direction
  //   and operation status
  absolute_force_vector.x = thruster_force.get_magnitude()
                            *
thruster_force.get_components().x;
  absolute_force_vector.y = thruster_force.get_magnitude()
                            *
thruster_force.get_components().y;

  absolute_force_vector.z = thruster_force.get_magnitude()
                            *
thruster_force.get_components().z;

  absolute_torque_vector.x = absolute_force_vector.y *
body_location.z +
                             absolute_force_vector.z *
body_location.y;
  absolute_torque_vector.y = absolute_force_vector.z *
body_location.x +
                             absolute_force_vector.x *
body_location.z;
  absolute_torque_vector.z = absolute_force_vector.y *
body_location.x +
                             absolute_force_vector.x *
body_location.y;
};
```

```
//////////////////////////////////////////////////////////////////////
//////
//
//  SubmarineVehicle
//
//////////////////////////////////////////////////////////////////////
//////

SubmarineVehicle:: SubmarineVehicle ()
  : OFF_Drawable_Obj() {
};

SubmarineVehicle:: SubmarineVehicle(char *f)
  : OFF_Drawable_Obj(f) {
};

SubmarineVehicle:: ~SubmarineVehicle() {};

void SubmarineVehicle:: add_weight (double w) {
  weight += w;
};

void SubmarineVehicle:: add_buoyancy (double b) {
  buoyancy += b;
};

void SubmarineVehicle:: set_weight (double w) {
  weight = w;
};

void SubmarineVehicle:: set_buoyancy (double b) {
  buoyancy = b;
};

void SubmarineVehicle:: set_sea_current (double uc, double
vc, double wc) {
  sea_current.x = uc;
  sea_current.y = vc;
  sea_current.z = wc;
};

double SubmarineVehicle:: get_weight () {
```

```
  return weight;
};

double SubmarineVehicle:: get_buoyancy () {
  return buoyancy;
};

//////////////////////////////////////////////////////////////////////
//
//  DSRV_obj
//
//////////////////////////////////////////////////////////////////////

DSRV_obj:: DSRV_obj()
  : SubmarineVehicle("off_files/dsrv.off"),
    ballast(500.0, 250.0, 50.0, 50.0) {

  // assign initial values from table
  //   ballast is ignored for mass quantity
  mass        = body_mass;
  weight      = body_weight + ballast.get_current_ballast();
  buoyancy    = body_boy + ballast.get_current_ballast();

  // set thruster performance values
  fwd_transverse_thruster = new Thruster;
  fwd_transverse_thruster->set_max_thrust(400.0);
  fwd_transverse_thruster->set_thruster_location (19.0, 0.0,
0.0);
  fwd_transverse_thruster->set_thruster_direction(0.0, 1.0,
0.0);
  fwd_transverse_thruster->set_thruster_rpm (100.0);
  fwd_transverse_thruster->set_screw_efficiency(0.2);

  aft_transverse_thruster = new Thruster;
  aft_transverse_thruster->set_max_thrust(400.0);
  aft_transverse_thruster->set_thruster_location (-19.0,
0.0, 0.0);
  aft_transverse_thruster->set_thruster_direction(0.0, 1.0,
0.0);
  aft_transverse_thruster->set_thruster_rpm (100.0);
  aft_transverse_thruster->set_screw_efficiency(0.2);
```

```
  fwd_vertical_thruster = new Thruster;
  fwd_vertical_thruster->set_max_thrust(400.0);
  fwd_vertical_thruster->set_thruster_location (21.0, 0.0,
0.0);
  fwd_vertical_thruster->set_thruster_direction(0.0, 0.0,
1.0);
  fwd_vertical_thruster->set_thruster_rpm (100.0);
  fwd_vertical_thruster->set_screw_efficiency(0.2);

  aft_vertical_thruster = new Thruster;
  aft_vertical_thruster->set_max_thrust(400.0);
  aft_vertical_thruster->set_thruster_location (-21.0, 0.0,
0.0);
  aft_vertical_thruster->set_thruster_direction(0.0, 0.0,
1.0);
  aft_vertical_thruster->set_thruster_rpm (100.0);
  aft_vertical_thruster->set_screw_efficiency(0.2);

  propulsor = new Propulsor;
  propulsor->set_max_thrust(89.0 * 0.75);
  propulsor->set_thruster_location (-25.0, 0.0, 0.0);
  propulsor->set_thruster_direction(1.0, 0.0, 0.0);
  propulsor->set_thruster_rpm (0.0);
  propulsor->set_screw_efficiency(0.75);

  u     =  u0;
  v     =  v0;
  w     =  w0;
  p     =  p0;
  q     =  q0;
  r     =  r0;
  xpos  =  xpos0;
  ypos  =  ypos0;
  zpos  =  zpos0;
  phi   =  phi0;
  theta =  theta0;
  psi   =  psi0;

  // set a starting position over and aft of the submarine
  xpos  = -100.0;
  zpos  =  850.0;
```

```
  // initial rudder and stern plane angles from table
  dr    = DEGRAD * dr;
  ds    = DEGRAD * dr;


    assign initial conditions to XX VECTOR
  xx[1]  = u;
  xx[2]  = v;
  xx[3]  = w;
  xx[4]  = p;
  xx[5]  = q;
  xx[6]  = r;
  xx[7]  = xpos;
  xx[8]  = ypos;
  xx[9]  = zpos;
  xx[10] = phi;
  xx[11] = theta;
  xx[12] = psi;

  // build the mass matrix
  M[1][1] =  mass - xudot;          M[1][5] =  mass*zg;
  M[1][6] = -mass*yg;

  M[2][2] =  mass - yvdot;          M[2][4] = -mass*zg -
ypdot;
  M[2][6] =  mass*xg - yrdot;

  M[3][3] =  mass - zwdot;          M[3][4] =  mass*yg;
  M[3][5] = -mass*xg - zqdot;

  M[4][2] = -mass*zg - kvdot;       M[4][3] =  mass*yg;
  M[4][4] =  Ixx - kpdot;           M[4][5] = -Ixy;
  M[4][6] = -Ixz - krdot;

  M[5][1] =  mass*zg;               M[5][3] = -mass*xg -
mwdot;
  M[5][4] = -Ixy;                   M[5][5] =  Iyy - mqdot;
  M[5][6] = -Iyz;

  M[6][1] = -mass*yg;               M[6][2] =  mass*xg -
nvdot;
  M[6][4] = -Ixz - npdot;           M[6][5] = -Iyz;
  M[6][6] =  Izz - nrdot;
```

```
  // prepare matrix for use by inverting
  invert_matrix (&M[0][0], &Mi[0][0], 6);

};

DSRV_obj:: ~DSRV_obj() {
  delete fwd_transverse_thruster;
  delete aft_transverse_thruster;
  delete fwd_vertical_thruster;
  delete aft_vertical_thruster;
  delete propulsor;

//  delete propeller;
};

void DSRV_obj:: set_rudder_angle(double r) {
  // increment rudder angle (in radians)
  dr += r;

  // rudder angle limits are +- 32 degrees
  if (dr < -32.0 * DEGRAD) dr = -DEGRAD * 32.0;
  if (dr >  32.0 * DEGRAD) dr =  DEGRAD * 32.0;
};

void DSRV_obj:: set_sternplane_angle(double s) {
  // increment stern plane angle (in radians)
  ds += s;

  // stern plane angle limits are +- 28 degrees
  if (ds < -28.0 * DEGRAD) ds = -DEGRAD * 28.0;
  if (ds >  28.0 * DEGRAD) ds =  DEGRAD * 28.0;
};

void DSRV_obj:: increment_propulsor_rpm (double r) {
  propulsor->increment_propulsor_rpm (r);
};

void DSRV_obj:: stop_propulsor () {
  propulsor->set_propulsor_rpm (0.0);
};
```

```
void DSRV_obj:: toggle_fwd_transverse_thruster (int
direction) {
  if (direction == 1) fwd_transverse_thruster-
>start_positive_thrust();
  else fwd_transverse_thruster->start_negative_thrust();
};

void DSRV_obj:: toggle_aft_transverse_thruster (int
direction) {
  if (direction == 1) aft_transverse_thruster-
>start_positive_thrust();
  else aft_transverse_thruster->start_negative_thrust();
};

void DSRV_obj:: toggle_fwd_vertical_thruster (int direction)
{
  if (direction == 1) fwd_vertical_thruster-
>start_positive_thrust();
  else fwd_vertical_thruster->start_negative_thrust();
};

void DSRV_obj:: toggle_aft_vertical_thruster (int direction)
{
  if (direction == 1) aft_vertical_thruster-
>start_positive_thrust();
  else aft_vertical_thruster->start_negative_thrust();
};

void DSRV_obj:: update() {

  // determine time interval for numerical integration
  float dt = delta_time();

  // adjust weight for ballast system operation
  ballast.update_ballast(dt);
  weight = body_weight + ballast.get_current_ballast();

  // adjust thruster internal states
  fwd_transverse_thruster->update_thruster_force(dt);
  aft_transverse_thruster->update_thruster_force(dt);
  fwd_vertical_thruster->update_thruster_force(dt);
  aft_vertical_thruster->update_thruster_force(dt);
```

```
  // adjust propulsor state
  propulsor->update_thruster_force(dt);

  // CALCULATE THE DRAG FORCE, INTEGRATE THE DRAG OVER THE
VEHICLE

    cf_flag = 0;
     for (int k=1;k<=15;++k)
     {
      ucf = pow( (v + x_cf[k]*r),2.0) + pow( (w -
x_cf[k]*q),2.0);
     ucf = sqrt(ucf);
      if (ucf < 1.0e-6)
      {
  cf_flag = 1;
  break;
     }

      cflow = cdy*hh_cf[k]*pow( (v + x_cf[k]*r),2.0) +
        cdz*br_cf[k]*pow( (w - x_cf[k]*q),2.0);
    vech1[k] = cflow*(v + x_cf[k]*r)/ucf;
   vech2[k] = cflow*(v + x_cf[k]*r)*x_cf[k]/ucf;
    vecv1[k] = cflow*(w - x_cf[k]*q)/ucf;
   vecv2[k] = cflow*(w - x_cf[k]*q)*x_cf[k]/ucf;
    }

    if (cf_flag == 0)
    {
     cf_heave = trapezoid_integration(15,vecv1,x_cf);
     cf_pitch = trapezoid_integration(15,vecv2,x_cf);
     cf_sway  = trapezoid_integration(15,vech1,x_cf);
     cf_yaw   = trapezoid_integration(15,vech2,x_cf);

     cf_heave = -0.5*rho*cf_heave;
     cf_pitch = 0.5*rho*cf_pitch;
     cf_sway  = -0.5*rho*cf_sway;
     cf_yaw   = -0.5*rho*cf_yaw;
    }
    else
    {
     cf_heave = 0.0;
     cf_pitch = 0.0;
     cf_sway  = 0.0;
```

```
        cf_yaw    = 0.0;
      }

      /* SURGE FORCE */

    suf1 = mass*(v*r - w*q + xg*(pow(q,2.0) + pow(r,2.0)) ) +
     mass*(-yg*p*q - zg*p*r) + xpp*pow(p,2.0) + xqq*pow(q,2.0)
     + xrr*pow(r,2.0) + xpr*p*r + xwq*w*q + xvp*v*p + xvr*v*r
     - (weight - buoyancy)*sin(theta) - xres*u*fabs(u);
      suf2 = u*q*(xqds*ds + xqdb*db) + u*r*(xrdrs*dr);
      suf3 = (xdsds*pow(ds,2.0) + xdbdb*pow(db,2.0)
     + xdrdr*(pow(dr,2.0)))*u*fabs(u);
      suf4 = propulsor->get_thrust().x;
      f[1] = suf1 + suf2 + suf3 + suf4;

      /* SWAY FORCE */

      swf1 = mass*(-u*r + w*p - xg*p*q + yg*(pow(p,2.0) +
pow(r,2.0))
     - zg*q*r)
     + (weight-buoyancy)*sin(phi)*cos(theta)
     + ypq*p*q + yqr*q*r + yp*u*p + yr*u*r + yvq*v*q + ywp*w*p
     + ywr*w*r + yv*u*v + yvw*v*w + cf_sway;
      swf2 = ydrs*u*fabs(u)*dr;
      f[2] = swf1 + swf2
           + fwd_transverse_thruster->get_thrust().y
           + aft_transverse_thruster->get_thrust().y;

      /* HEAVE FORCE */

      hf1 = mass*(u*q - v*p - xg*p*r - yg*q*r + zg*(pow(p,2.0)
+ pow(q,2.0)))
     + (weight - buoyancy)*cos(phi)*cos(theta) + zpp*pow(p,2.0)
+ zpr*p*r
     + zrr*pow(r,2.0) + zq*u*q + zvp*v*p + zvr*v*r + zw*u*w
     + zvv*pow(v,2.0) + cf_heave;
      hf2 = u*fabs(u) * zds * ds;
      f[3] = hf1 + hf2
           + fwd_vertical_thruster->get_thrust().z
           + aft_vertical_thruster->get_thrust().z;

      /* ROLL MOMENT */
```

```
    rm1 = -(Izz - Iyy)*q*r - Ixy*p*r + Iyz*(pow(q,2.0)
  - pow(r,2.0))+Ixz*p*q
  + mass*(yg*(u*q - v*p) - zg*(-u*r + w*p))
  + (yg*weight-yb*buoyancy)*cos(phi)*cos(theta)
  - (zg*weight-zb*buoyancy)*sin(phi)*cos(theta)
  + kpq*p*q + kqr*q*r + kp*u*p + kr*u*r + kvq*v*q + kwp*w*p
  + kwr*w*r + kv*u*v + kvw*v*w - 100.0*p*fabs(p);

    f[4] = rm1;

    /* PITCH MOMENT */

    pm1 = -(Ixx - Izz)*p*r + Ixy*q*r - Iyz*p*q
  - Ixz*(pow(p,2.0)-pow(r,2.0))
  + mass*( xg*(-u*q + v*p) + zg*(v*r - w*q) )
  - (xg*weight - xb*buoyancy)*cos(phi)*cos(theta)
  - (zg*weight - zb*buoyancy)*sin(theta)
  + mpp*pow(p,2.0) + mpr*p*r + mrr*pow(r,2.0) + mq*u*q +
mvp*v*p
  + mvr*v*r + mw*u*w + mvv*pow(v,2.0) + cf_pitch;
    pm2 = u*fabs(u) * mds*ds;
    f[5] = pm1 + pm2
         + fwd_vertical_thruster->get_torque().y
         + aft_vertical_thruster->get_torque().y;

    /* YAW MOMENT */

    ym1 = -(Iyy - Ixx)*p*q + Ixy*(pow(p,2.0) - pow(q,2.0)) +
Iyz*p*r
  - Ixz*q*r
  + mass*(xg*(w*p - u*r) + yg*(w*q-v*r))
  + (xg*weight-xb*buoyancy)*sin(phi)*cos(theta)
  + (yg*weight-yb*buoyancy)*sin(theta)
  + npq*p*q + nqr*q*r + np*u*p + nr*u*r + nvq*v*q + nwp*w*p
  + nwr*w*r + nv*u*v + nvw*v*w +  cf_yaw;
    ym2 = ndrs*u*fabs(u)*dr;
    f[6] = ym1 + ym2
         + fwd_transverse_thruster->get_torque().z
         + aft_transverse_thruster->get_torque().z;

    /* INERTIAL POSITION RATES */

    s_phi = sin(phi); s_theta = sin(theta); s_psi = sin(psi);
```

```
    c_phi = cos(phi); c_theta = cos(theta); c_psi = cos(psi);
    t_theta = tan(theta);

  f[7] = sea_current.x + u*c_psi*c_theta;
    f[7] = f[7] + v*(c_psi*s_theta*s_phi - s_psi*c_phi);
    f[7] = f[7] + w*(c_psi*s_theta*c_phi + s_psi*s_phi);

  f[8] = sea_current.y + u*s_psi*c_theta;
    f[8] = f[8] + v*(s_psi*s_theta*s_phi + c_psi*c_phi);
    f[8] = f[8] + w*(s_psi*s_theta*c_phi - c_psi*s_phi);

  f[9] = sea_current.z - u*s_theta + v*c_theta*s_phi;
    f[9] = f[9] + w*c_theta*c_phi;

    /* EULER ANGLE RATES */

    f[10]= p + q*s_phi*t_theta + r*c_phi*t_theta;

    f[11]= q*c_phi - r*s_phi;

    f[12]= (1/c_theta)*(q*s_phi + r*c_phi);

    /* BUILD STATE MATRIX AND COMPUTE THE RIGHT HAND SIDE OF
XDOT=F(X)
     XXDOT = [imm zeros(6,6);zeros(6,6) eye(6,6)]*f' */

    for (int j=1;j<=12;++j)
    {
     for (k=1;k<=12;++k)
     {
  state_matrix[j][k] = 0.0;
     }
    }

    for (j=1;j<=6;++j)
    {
     for (k=1;k<=6;++k)
     {
  state_matrix[j][k] = Mi[j][k];
     }
    }

    for (j=7;j<=12;++j)
```

```
    {
     state_matrix[j][j] = 1.0;
    }

    for (j=1;j<=12;++j)
    {
     xxdot[j] = 0.0;
     for (k=1;k<=12;++k)
     {
  xxdot[j] = xxdot[j] + state_matrix[j][k]*f[k];
     }
    }

    for (j=1;j<=12;++j)
    {
     xx[j] = xx[j] + dt * xxdot[j];
    }

  printf("%f %f %f %f %f %f %f %f \n",xpos, ypos, zpos,
phi*RADDEG, theta*RADDEG, psi*RADDEG, n_rpm, dr*RADDEG);

  // update state values
  u     = xx[1];
  v     = xx[2];
  w     = xx[3];
  p     = xx[4];
  q     = xx[5];
  r     = xx[6];
  xpos  = xx[7];
  ypos  = xx[8];
  zpos  = xx[9];
  phi   = xx[10];
  theta = xx[11];
  psi   = xx[12];

  // update state values
  vel.x     = xx[1];
  vel.y     = xx[2];
  vel.z     = xx[3];
  ang_vel.x = xx[4];
  ang_vel.y = xx[5];
  ang_vel.z = xx[6];
  pos.x     = xx[7];
```

```
  pos.y      = xx[8];
  pos.z      = xx[9];
  orient.x   = xx[10];
  orient.y   = xx[11];
  orient.z   = xx[12];

  // dummy stub to show where network update calls
  // belong.
//  network_player.computeNextState(pos.x, pos.y, pos.z);
//  network_ghost.computeNextState(dt);

};

DSRV_obj:: DSRV_obj(char * f)
  : SubmarineVehicle(f) {
};

void DSRV_obj:: Draw() {

  // draw the DSRV
  pushmatrix();
    translate(pos.x, pos.y, pos.z);
    rot(-90, 'X');
    rot(-orient.z*RADDEG, 'Y');
    rot(orient.y*RADDEG, 'Z');
    rot(orient.x*RADDEG, 'X');
    display_this_object(image);
  popmatrix();

};


int DSRV_obj:: ballast_pump_is_on() {
  return ballast.pump_is_on();
};

int DSRV_obj:: sea_valve_is_open() {
  return ballast.sea_valve_is_open();
};


/* function invert_matrix(a,ai,n):
   where a  = n x n matrix
```

```
         ai = n x n inverse of matrix a
         n  = row & column dimension of matrix a
    Usage:
         .
         .
         inv(&a,&ai,n);
         .
         .

   NOTE: Matrices in calling program must be dimensioned
         a[n+1][n+1], ai[n+1][n+1]. If n > 30 array size
below
         must be increased.
*/
//void inv(double *a, double *ai, int n)
//void inv(double *a[][7], double *ai[][7], int n)

void DSRV_obj:: invert_matrix(double *a, double *ai, int n)
{

  int ki;
  double b,b1,b2;
  double a_local[30][30],ainv[30][30];

  int sing_flag = 0;

  for(int i=1; i<=n; ++i)
  {
     for(int j=1; j<=n; ++j)
     {
  a_local[i][j] =  *(a+i*(n+1)+j);
     }
  }

  for (i=1; i<=n; ++i)
  {
     for ( int j=1; j<=n; ++j)
     {
  ainv[i][j] = 0.0;
     }
  }

  for (i=1; i<=n; ++i)
```

```
{
   ainv[i][i] = 1.0;
}

for (int k=1; k<=n-1; ++k)
{
   b = a_local[k][k];
   ki = k;

   for (i=k+1; i<=n; ++i)
   {
if( (fabs(b) - fabs(a_local[i][k])) >= 0.0 )
{
}
else
{
   b = a_local[i][k];
   ki = i;
}
   }

   if( fabs(b) < 0.0001)
   {
sing_flag = 1;
break;
   }

   if( (ki-k) == 0)
   {
   }
   else
   {
for (int j=k; j<=n; ++j)
{
   b1 = a_local[k][j];
   a_local[k][j] = a_local[ki][j];
   a_local[ki][j] = b1;
}
for (j=1; j<=n; ++j)
{
   b2 = ainv[k][j];
   ainv[k][j] = ainv[ki][j];
   ainv[ki][j] = b2;
```

```
  }
     }

     for (int j=k+1; j<=n; ++j)
     {
  a_local[k][j] = a_local[k][j]/b;
     }

     for (j=1; j<=n; ++j)
     {
  ainv[k][j] = ainv[k][j]/b;
     }

     for (i=k+1; i<=n; ++i)
     {
  for (j=k+1; j<=n; ++j)
  {
     a_local[i][j] = a_local[i][j] -
a_local[i][k]*a_local[k][j];
  }

  for (j=1; j<=n; ++j)
  {
     ainv[i][j] = ainv[i][j] - a_local[i][k]*ainv[k][j];
  }
     }
  }

  if(sing_flag == 0)
  {
     for (int j=1; j<=n; ++j)
     {
  ainv[n][j] = ainv[n][j]/a_local[n][n];
     }

     for (int k=n-1; k>=1; --k)
     {
  for (int j=1; j<=n; ++j)
  {
     for (int i=k+1; i<=n; ++i)
     {
        ainv[k][j] = ainv[k][j] - a_local[k][i]*ainv[i][j];
     }
```

```
  }
     }

     for(int i=1; i<=n; ++i)
     {
  for(int j=1; j<=n; ++j)
  {
     *(ai+i*(n+1)+j) = ainv[i][j];
  }
     }
  }
  else
  {
//      cout << "Singular or Ill-Conditioned Matrix\n";
  }
};


// Numerical integration routine using the trapezoidal rule
double DSRV_obj:: trapezoid_integration (int n, double a[],
double b[]) {
   double out1;

   int    n1  = n - 1;
   double out = 0.0;

   for (int i=1; i<=n1; ++i)
   {
      out1 = 0.5*(a[i] + a[i+1])*(b[i+1] - b[i]);
      out  = out + out1;
   }
   return out;
};

////////////////////////////////////////////////////////////////
//////////
//
//  Submarine_obj
//
////////////////////////////////////////////////////////////////
//////////

Submarine_obj:: Submarine_obj()
```

```
  : SubmarineVehicle("off_files/my688.off") {
  pos.z = 987.0;
};

Submarine_obj:: Submarine_obj(char *f)
  : SubmarineVehicle(f) {
  propeller = read_object ("off_files/prop.off");

   // does propeller exist?
   if(propeller == (OBJECT *) NULL)
   {
      printf("Illegal off file specification\n");

   }

//  ready_object_for_display (propeller),
  prop_rot = 0.0;

};

void DSRV_obj:: set_image (OBJECT* i) {
  image = i;
};

OBJECT* DSRV_obj:: get_image () {
  return image;
};

Submarine_obj:: ~Submarine_obj() {
//  delete propeller;
};

void Submarine_obj:: update() {
};

void Submarine_obj:: Draw() {

  // draw stricken submarine
  pushmatrix();
    translate(pos.x, pos.y, pos.z);
    rot(-90, 'X');
    rot(orient.z, 'Y');
    scale(2.0, 2.0, 2.0);
```

```
    display_this_object(image);
  popmatrix();
};

void Submarine_obj:: set_image (OBJECT* i) {
  image = i;
};

OBJECT* Submarine_obj:: get_image () {
  return image;
};
```

APPENDIX D

DYNAMIC OBJECTS

```
#ifndef _DYNAMIC_OBJS
#define _DYNAMIC_OBJS

#define PI     3.1415927
#define RADDEG 57.29577951
#define DEGRAD 0.017453293

struct ThreeVector {

    double x;
    double y;
    double z;
};

struct ThreeMatrix {

    ThreeVector x;
    ThreeVector y;
    ThreeVector z;
};

class Motive {

  public:
    Motive();
    Motive(double);
    Motive(double, double, double);
    ~Motive();

    virtual void        set_magnitude(double);
    virtual void        set_components(double, double,
double);
    virtual double      get_magnitude();
    virtual ThreeVector get_components();
```

```
  protected:
    double      mag;
    ThreeVector comp;
};

class Force: public Motive {

  public:
    Force();
    Force(double);
    Force(double, double, double);
    ~Force();

    void set_magnitude  (double);
    void set_components (double, double, double);

    double      get_magnitude();
    ThreeVector get_components();
};

class Torque: public Motive {

  public:
    Torque();
    Torque(double);
    Torque(double, double, double);
    ~Torque();

    void        set_magnitude  (double);
    void        set_components (double, double, double);
    double      get_magnitude();
    ThreeVector get_components();
};


class PointMass: public Force {

  public:
    PointMass();
    PointMass(double);
    PointMass(double, double, double, double);
    ~PointMass();
```

```
    void        set_mass    (double);
    void        set_position(double, double, double);
    void        set_velocity(double, double, double);
    void        set_acceleration(double, double, double);

    double      get_mass  ();
    ThreeVector get_pos   ();
    ThreeVector get_vel   ();
    ThreeVector get_accel ();
    long        get_time  ();

    virtual void fwd_kinematics     (long) = 0;
    virtual void fwd_dynamics       (long) = 0;
    virtual void reverse_kinematics ()     = 0;
    virtual void reverse_dynamics   ()     = 0;
    virtual void update             ()     = 0;

  protected:
    double      mass;
    ThreeVector pos;
    ThreeVector vel;
    ThreeVector accel;
    long        time;

    float       delta_time();
};

class RigidBody: public PointMass, public Torque {

  public:
    RigidBody();
    ~RigidBody();

    void set_inertia   (ThreeVector, ThreeVector,
ThreeVector);
    void set_orient    (double, double, double);
    void set_ang_vel   (double, double, double);
    void set_ang_accel (double, double, double);

    ThreeMatrix get_inertia   ();
    ThreeVector get_orient    ();
    ThreeVector get_ang_vel   ();
    ThreeVector get_ang_accel ();
```

```
    virtual void fwd_kinematics     (long);
    virtual void fwd_dynamics       (long);
    virtual void reverse_kinematics ();
    virtual void reverse_dynamics   ();
    virtual void update();


  protected:
    ThreeMatrix inertia;
    ThreeVectororient;
    ThreeVectorang_vel;
    ThreeVector ang_accel;
};

#endif


//////////////////////////////////////////////////////////////////
//
//  Dynamic_Objs.C
//
//////////////////////////////////////////////////////////////////

#include <math.h>
#include <iostream.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/param.h>
#include "Dynamic_Objs.H"

#define RADDEG 57.29577951
#define DEGRAD 0.017453293
#define GRAV 9.82


Motive:: Motive() {
  mag  = 0.0;
  comp.x = 0.0; comp.y = 0.0; comp.z = 0.0;
};

Motive:: Motive(double m)
  : mag(m) {
```

```
  comp.x = 0.0; comp.y = 0.0; comp.z = 0.0;
};

Motive:: Motive (double a, double b, double c)
  : mag(0.0) {
  comp.x = a; comp.y = b; comp.z = c;
};

Motive:: ~Motive() {};

void Motive:: set_magnitude(double m) {
  mag = m;
};

void Motive:: set_components(double a, double b, double c) {
  comp.x = a; comp.y = b; comp.z = c;
};

double Motive:: get_magnitude() {
  return mag;
};

ThreeVector Motive:: get_components() {
  return comp;
};

Force:: Force()
  : Motive () {
};

Force:: Force(double m)
  : Motive (m) {
};

Force::Force(double a, double b, double c)
  : Motive(a, b, c) {
};

Force::~Force() {};

void Force:: set_magnitude (double m) {
  Motive::set_magnitude(m);
};
```

```
void Force:: set_components (double a, double b, double c) {
  Motive::set_components (a, b, c);
};

double Force:: get_magnitude() {
  return mag;
};

ThreeVector Force:: get_components() {
  return comp;
};

Torque:: Torque() {};

Torque:: Torque(double m)
  : Motive(m) {
};

Torque:: Torque (double a, double b, double c)
  : Motive(a, b, c) {
};

Torque::~Torque() {};

void Torque:: set_magnitude (double m) {
  Motive::set_magnitude(m);
};

void Torque:: set_components (double a, double b, double c)
{
  set_components (a, b, c);
};

double Torque:: get_magnitude() {
  return mag;
};

ThreeVector Torque:: get_components() {
  return comp;
};

PointMass:: PointMass() {
```

```
  struct tms time_buff;

  mass     = 0.0;
  pos.x    = 0.0; pos.y    = 0.0; pos.z    = 0.0;
  vel.x    = 0.0; vel.y    = 0.0; vel.z    = 0.0;
  accel.x  = 0.0; accel.y  = 0.0; accel.z  = 0.0;
  time     = times(&time_buff);
};

PointMass:: PointMass(double m) {
  struct tms time_buff;

  mass     = m;
  pos.x    = 0.0; pos.y    = 0.0; pos.z    = 0.0;
  vel.x    = 0.0; vel.y    = 0.0; vel.z    = 0.0;
  accel.x  = 0.0; accel.y  = 0.0; accel.z  = 0.0;
  time     = times(&time_buff);
};

PointMass:: PointMass(double m,
                      double px, double py, double pz) {
  struct tms time_buff;

  mass     = m;
  pos.x    = px; pos.y     = py;  pos.z    = pz;
  vel.x    = 0.0; vel.y    = 0.0; vel.z    = 0.0;
  accel.x  = 0.0; accel.y  = 0.0; accel.z  = 0.0;
  time     = times(&time_buff);
};

PointMass:: ~PointMass() {};

void PointMass:: set_mass(double m) {
  mass = m;
};

void PointMass:: set_position(double px, double py, double
pz) {
  pos.x = px;
  pos.y = py;
  pos.z = pz;
};
```

```
void PointMass:: set_velocity(double vx, double vy, double
vz) {
  vel.x = vx;
  vel.y = vy;
  vel.z = vz;
};

void PointMass:: set_acceleration(double ax, double ay,
double az) {
  accel.x = ax;
  accel.y = ay;
  accel.z = az;
};

double PointMass:: get_mass() {
  return mass;
};

ThreeVector PointMass:: get_pos() {
  return pos;
};


ThreeVector PointMass:: get_vel() {
  return vel;
};

ThreeVector PointMass:: get_accel() {
  return accel;
};

long PointMass:: get_time() {
  return time;
};

float PointMass:: delta_time () {
  struct tms time_buff;
  long       temp = time;

  time = times (&time_buff);
  return (float)(time - temp) / HZ;
};
```

```
RigidBody:: RigidBody() {
  orient.x    = 0.0; orient.y    = 0.0; orient.z    = 0.0;
  ang_vel.x   = 0.0; ang_vel.y   = 0.0; ang_vel.z   = 0.0;
  ang_accel.x = 0.0; ang_accel.y = 0.0; ang_accel.z = 0.0;
};

RigidBody:: ~RigidBody() {};

void RigidBody:: set_inertia(ThreeVector ix, ThreeVector iy,
ThreeVector iz) {
  inertia.x = ix;
  inertia.y = iy;
  inertia.z = iz;
};

void RigidBody:: set_orient(double ox, double oy, double oz)
{
  orient.x = ox;
  orient.y = oy;
  orient.z = oz;
};

void RigidBody:: set_ang_vel(double vx, double vy, double
vz) {
  ang_vel.x = vx;
  ang_vel.y = vy;
  ang_vel.z = vz;
};

void RigidBody:: set_ang_accel(double ax, double ay, double
az) {
  ang_accel.x = ax;
  ang_accel.y = ay;
  ang_accel.z = az;
};

ThreeMatrix RigidBody:: get_inertia() {
  return inertia;
};

ThreeVector RigidBody:: get_orient() {
  return orient;
};
```

```
 ThreeVector RigidBody:: get_ang_vel() {
  return ang_vel;
};

ThreeVector RigidBody:: get_ang_accel() {
  return ang_accel;
};

void RigidBody:: fwd_kinematics(long) {};

void RigidBody:: fwd_dynamics(long) {};

void RigidBody:: reverse_kinematics() {};

void RigidBody:: reverse_dynamics() {};

void RigidBody:: update() {};
```

APPENDIX E

SIMULATION WINDOW

```
#include "Window.H"
#include "Sub_Vehicle_Objs.H"

enum viewSelect {TOPVIEW, SIDEVIEW, DSRVVIEW};
enum boolean    {ON, OFF};

class SimulationWindow : public Window {

  public:
    SimulationWindow ();
    ~SimulationWindow();

    void PassDrawObj(DSRV_obj *, Submarine_obj *);

    // define inherited functions from class Window
    void evaluate_menu_selection();
    void DrawWindow ();

  private:
    viewSelect       viewselection;
    boolean          selectgrid;

    DSRV_obj         *dsrv;
    Submarine_obj    *sub;
    OBJECT           *light_obj; // OFF objects
    OBJECT           *ref_obj;
    OBJECT           *floor;

    // defines variabels for using fogverex commands to
    //   simulate obscured underwater vision
    static float     fog[5];


    // define inherited functions from class Window
    long makethemenus();
    void processmenuhit();
```

```
};


//////////////////////////////////////////////////////////////
//
//  Simulation_Window.C
//
//////////////////////////////////////////////////////////////

#include <gl.h>
#include "Simulation_Window.H"
#include "Sub_Vehicle_Objs.H"

#define NEARCLIPPING 0.1
#define FARCLIPPING  2000.0

SimulationWindow:: SimulationWindow() {

  viewselection = TOPVIEW;
  selectgrid    = ON;

//  topview.set_field_of_view(450);
//  topview.set_aspect(1.25);
//  topview.set_polar_view(300.0, 0, 1800, 0);

//  sideview.set_field_of_view(900);
//  sideview.set_aspect(1.25);

//  dsrvview.set_field_of_view (600);
//  dsrvview.set_aspect(1.25);

  // define references for OFF file objects
  light_obj = read_object ("off_files/
underwater_light.off");
  ref_obj   = read_object ("off_files/reference.off");
  floor     = read_object ("off_files/seabottom.off");

  // OFF function call to prepare/define the visual objects
  ready_object_for_display (light_obj);
  ready_object_for_display (ref_obj);
  ready_object_for_display (floor);

  // make the popup menus for reference
```

```
  mainmenu = makethemenus();

};

SimulationWindow:: ~SimulationWindow() {};

void SimulationWindow:: PassDrawObj (DSRV_obj *d,
Submarine_obj *s) {
  dsrv = d;
  sub  = s;
};

void SimulationWindow:: evaluate_menu_selection() {

  // must be in MSINGLE mode to do popup menus
  mmode(MSINGLE);

  // which popup selection ?
  hititem = dopup(mainmenu);

  // put us back into MVIEWING mode
  mmode(MVIEWING);

  // do something with the popup hit
  processmenuhit();
};

void SimulationWindow:: DrawWindow() {

  // activate selected viewpoint
  switch (viewselection) {

    case TOPVIEW:

      // draw the background color
      czclear(0xFFFF7200,getgdesc(GD_ZMAX));

      // must do this in Mviewing
      loadmatrix (unit);

      // build the viewing matrix
      perspective(450,1.25,NEARCLIPPING,FARCLIPPING);
      polarview(-300.0, 0, 1800, 0);
```

```
      // turn off fog when not looking through dsrv viewport
      fogvertex(FG_OFF, fog);

      break;

    case SIDEVIEW:

      // draw the background color
      czclear(0xFFFF7200,getgdesc(GD_ZMAX));

      // build the viewing matrix
      perspective(900,1.25,NEARCLIPPING,FARCLIPPING);

      // must do this in Mviewing
      loadmatrix (unit);

      lookat(sub->get_pos().x, sub->get_pos().y + 200.0f,
sub->get_pos().z - 50.0f,
            sub->get_pos().x, sub->get_pos().y,          sub-
>get_pos().z - 50.0f, 0);
      // turn off fog when not looking through dsrv viewport
      fogvertex(FG_OFF, fog);

      break;

    case DSRVVIEW:

      // draw the background color
      czclear(0xFF773333,getgdesc(GD_ZMAX));

      // must do this in Mviewing
      loadmatrix (unit);

      // build the viewing matrix*/
      perspective(600,1.25,NEARCLIPPING,FARCLIPPING);
      lookat(dsrv->get_pos().x,      dsrv->get_pos().y, dsrv-
>get_pos().z + 10.0f,
            dsrv->get_pos().x + 50, dsrv->get_pos().y, dsrv-
>get_pos().z + 100.0f,
             (short)dsrv->get_orient().z * 10 + 900);

      // turn on underwater visual impairment
```

```
      fogvertex (FG_VTX_LIN, fog);
      fogvertex (FG_ON, fog);

      break;

    default:
      polarview(100.0, 0, 0, 0);
      break;
  }

  if (selectgrid == ON) {
    RGBcolor(255,0,0);
    move(dsrv->get_pos().x,          dsrv->get_pos().y, dsrv-
>get_pos().z);
    draw(dsrv->get_pos().x + 100.0, dsrv->get_pos().y, dsrv-
>get_pos().z);
    RGBcolor(0,255,0);
    move(dsrv->get_pos().x, dsrv->get_pos().y,         dsrv-
>get_pos().z);
    draw(dsrv->get_pos().x, dsrv->get_pos().y+ 100.0, dsrv-
>get_pos().z);
    RGBcolor(0,0,255);
    move(dsrv->get_pos().x, dsrv->get_pos().y, dsrv-
>get_pos().z);
    draw(dsrv->get_pos().x, dsrv->get_pos().y, dsrv-
>get_pos().z + 100.0);
  }

  // Draw the light
  display_this_object(light_obj);

  // draw the ocean floor
  pushmatrix();
    translate(0.0, 0.0, 1000.0);
    rot(-90, 'X');
    scale(10.0, 10.0, 10.0);
    display_this_object(floor);
  popmatrix();

  // draw the reference
  pushmatrix();
    translate(0.0, -200.0, 1000.0);
    rot(-90, 'X');
```

```
    scale(10.0, 10.0, 10.0);
    display_this_object(ref_obj);
  popmatrix();

  // draw the DSRV and submarine
  dsrv->Draw();
  sub->Draw();

  // pump and sea valve status
  ortho2 (100.0, XMAXSCREEN, 100.0, YMAXSCREEN);
  if (dsrv->ballast_pump_is_on()) {
    RGBcolor (255, 0, 0);
    cmov2 (20.0, YMAXSCREEN - 10.0);
    charstr ("Ballast Pump Running");
   }

  if (dsrv->sea_valve_is_open()) {
    RGBcolor (255, 0, 0);
    cmov2 (40.0, YMAXSCREEN - 10.0);
    charstr ("Sea Valve Open");
   }


  // change out back frame buffer with front frame buffer
  swapbuffers();

};

long SimulationWindow:: makethemenus() {

  // this routine performs all the menu construction call

  long topmenu;       // top level menu's name
  long view_menu;     // viewpoint selection menu name
  long grid_menu;     // reference grid for DSRV
  long current_menu; // toggle ocean current

  // define low level menus
  current_menu = defpup("Toggle Ocean Current %t| Current ON
%x7| Current OFF %8");
  grid_menu    = defpup("DSRV Refernce Grid %t| Grid ON %x5|
Grid OFF %x6");
```

```
  view_menu     = defpup("Viewpoint Selection %t| Top View
%x2| Side View %x3| DSRV View %x4");

  // define top level menu
  topmenu       = defpup("DSRV Main Menu %t | Select View %m|
DSRV Reference Grid %m| Toggle Ocean Current %m |Press ESC to
exit %x99",
                    view_menu, grid_menu, current_menu);

  return(topmenu);
};

void SimulationWindow:: processmenuhit() {

  switch(hititem) {

    case -1:
      // no selection, just return
      break;

    case 1:
      break;

    case 2:

      // Top view viewpoint selection
      viewselection = TOPVIEW;
      break;

    case 3:

      // Side view selected
      viewselection = SIDEVIEW;
      break;

    case 4:

      // DSRV view selected
      viewselection = DSRVVIEW;
      break;

    case 5:
```

```
      // DSRV reference grid set ON
      selectgrid = ON;
      break;

    case 6:

      // DSRV reference grid set ON
      selectgrid = OFF;
      break;

    case 7:

      // set ocean current velocity vector
      // note: the sub is not affected by current, on bottom!
      dsrv->set_sea_current(0.5, 0.5, 0.0);
      break;

    case 8:

      // turn current off
      dsrv->set_sea_current(0.0, 0.0, 0.0);
      break;

    case 99:
      break;

    default:
      break;

  } // end switch
};

float SimulationWindow:: fog[] = {110.0, 600.0, 0.21, 0.20,
0.6};
```

APPENDIX F

WINDOW

```
#include <gl.h>
#include "Dynamic_Objs.H"
#include "rdobj_opcodes.H"
#include "image_types.H"

#ifndef _WINDOWH
#define _WINDOWH

extern "C"
{
  extern OBJECT* read_object(char*);
  extern void ready_object_for_display(OBJECT*);
  extern void display_this_object(OBJECT*);
};

const static Matrix unit = { 1.0, 0.0, 0.0, 0.0,
                             0.0, 1.0, 0.0, 0.0,
                             0.0, 0.0, 1.0, 0.0,
                             0.0, 0.0, 0.0, 1.0 };

class Window {
  public:
    Window();
    ~Window();

    void set_window_title(char *);
    virtual void evaluate_menu_selection() = 0;
    virtual void DrawWindow() = 0;

  protected:
    char *win_title;
    long win_x_size;      // window size in pixels
    long win_y_size;
    long win_x1_pos;      // window position in pixels
    long win_x2_pos;
    long win_y1_pos;
```

```
    long win_y2_pos;
    long win_x_aspect;    // value 1..32767 (0x7fff)
    long win_y_aspect;
    long win_near_z;      // z-buffer clipping plane values
    long win_far_z;
    long mainmenu;        // popup menu's name
    long hititem;         // the item selected by the menu

    virtual long makethemenus ();  // sub class redefine to
be meaningfull
    virtual void processmenuhit () = 0;

};

class View {

  public:
    View();
    View(unsigned long, long);
    ~View();

    void set_eyepoint  (long, long, long);
    void set_lookpoint (long, long, long);
    void set_rotation  (short);
    void set_field_of_view(short);
    void set_aspect    (double);
    void set_clipping  (long, long);
    void set_zbuffer_depth (long);
    void set_polar_view(double, short, short, short);

    virtual void DrawView ();
    virtual void DrawPolarView();

  protected:

    ThreeVector   from;
    ThreeVector   to;
    short         rotation;
    short         field_of_view;
    double        aspect;
    long          near_clip; // clipping plane values
    long          far_clip;
    long          z_depth;   // z_buffer depth to clear
```

```
    unsigned long color_val; // color value to clear the
screen
    double        polar_dist;
    short         polar_azim;
    short         polar_incline;
    short         polar_twist;

};


class Drawable_Obj {

  public:
    Drawable_Obj() {};
    ~Drawable_Obj() {};

    virtual void Draw () = 0;

};

class OFF_Drawable_Obj {

  public:
    OFF_Drawable_Obj ();
    OFF_Drawable_Obj (char *);
    ~ OFF_Drawable_Obj ();

    void ready_OFF_Obj();
    virtual void Draw ();

  protected:
    char   *filename;
    OBJECT *image;
};

#endif


////////////////////////////////////////////////////////////
//
//  Window.C
//
////////////////////////////////////////////////////////////
```

```
#include <stdio.h>
#include "Window.H"

extern "C"
{
  extern OBJECT* read_object(char*);
  extern void ready_object_for_display(OBJECT*);
  extern void display_this_object(OBJECT*);
};


Window:: Window() {

  // set window default values
  win_title     = "Window";
  win_x_size    = XMAXSCREEN - 100; //XMAXSCREEN is a system
variable
  win_y_size    = YMAXSCREEN - 100;
  win_x1_pos    = 100;              //window position
  win_x2_pos    = XMAXSCREEN;
  win_y1_pos    = 100;
  win_y2_pos    = YMAXSCREEN;
  win_x_aspect  = XMAXSCREEN - 100; //window aspect ration
  win_y_aspect  = YMAXSCREEN - 100;
  win_near_z    = 0x000000;       //near and far planes used
for Zbuffering
  win_far_z     = 0x7fffff;

  // set a default window size
  prefsize (win_x_size, win_y_size);

  // set a default window position
  prefposition (win_x1_pos, win_x2_pos, win_y1_pos,
win_y2_pos);

  // set a default aspect window ratio
  keepaspect (win_x_aspect, win_y_aspect);

  // open window with title
  winopen (win_title);

  // put IRIS in double buffer mode
```

```
  doublebuffer();

  // put IRIS in RGB mode
  RGBmode();

  // this call sets the settings established above
  gconfig();

  // set z-buffering depth
  lsetdepth (win_near_z, win_far_z);

  // set the gouraud shade model.  GOURAUD is a gl library
variable
  shademodel (GOURAUD);

  // enable the new projection matrix. MVIEWING is a gl
library variable
  mmode (MVIEWING);

  // turn on z-buffering. TRUE is a predefined library
variable
  zbuffer (TRUE);

  // turn the cursor on
  curson();

  // make the popup menus
//  mainmenu = makethemenus();
  hititem  = 0;
};

Window:: ~Window() {};

void Window:: set_window_title (char *t) {
  *win_title = *t;
};

long Window:: makethemenus () {

  // this routine is an example of the menu construction
calls

  long topmenu;       // top level menu's name
```

```
  long sub_menu;      // viewpoint selection menu name

  // define low level menus
  sub_menu = defpup("Sub menu %t| Sub menu selection %x1");

  // define top level menu
  topmenu       = defpup("Main Menu Sample %t | Selection
submenu %m|Exit %x99",
                         sub_menu);

  return(topmenu);
};

View:: View() {
  from.x = 0.0; from.y = 0.0; from.z = 0.0;
  to.x   = 0.0; to.y   = 0.0; to.z   = 0.0;
  rotation      = 0;
  field_of_view = 900;
  aspect        = 1.25;
  near_clip     = 0.1;
  far_clip      = 2000.0;
  z_depth       = getgdesc(GD_ZMAX);  // get system max z-
buffer depth value
  color_val     = 0xFFFF7200;
  polar_dist    = 0.0;
  polar_azim    = 0;
  polar_incline = 0;
  polar_twist   = 0;
};

View:: View(unsigned long c, long z) {
  from.x = 0.0; from.y = 0.0; from.z = 0.0;
  to.x   = 0.0; to.y   = 0.0; to.z   = 0.0;
  rotation      = 0;
  field_of_view = 900;
  aspect        = 1.25;
  near_clip     = 0.1;
  far_clip      = 2000.0;
  z_depth       = z;
  color_val     = c;
  polar_dist    = 0.0;
  polar_azim    = 0;
  polar_incline = 0;
```

```
  polar_twist    = 0;

};

View:: ~View() {};

void View:: set_eyepoint (long e1, long e2, long e3) {
  from.x = e1; from.y = e2; from.z = e3;
};

void View:: set_lookpoint(long l1, long l2, long l3) {
  to.x = l1; to.y = l2; to.z = l3;
};

void View:: set_rotation (short r) {
  rotation = r;
};

void View:: set_field_of_view(short f) {
  field_of_view = f;
};

void View:: set_aspect(double a) {
  aspect = a;
};

void View:: set_clipping (long n, long f) {
  near_clip = n;
  far_clip  = f;
};

void View:: set_zbuffer_depth (long d) {
  z_depth = d;
};

void View:: set_polar_view (double d, short a, short i,
short t) {
  polar_dist    = d;
  polar_azim    = a;
  polar_incline = i;
  polar_twist   = t;
};
```

```
void View:: DrawView () {
  czclear (color_val, z_depth);
  perspective (field_of_view, aspect, near_clip, far_clip);
  loadmatrix(unit);
  lookat (from.x, from.y, from.z,
          to.x,   to.y,   to.z,   rotation);
};

void View:: DrawPolarView () {
  perspective (field_of_view, aspect, near_clip, far_clip);
  polarview   (polar_dist, polar_azim, polar_incline,
polar_twist);
};

OFF_Drawable_Obj:: OFF_Drawable_Obj() {
};

OFF_Drawable_Obj:: OFF_Drawable_Obj (char *f) {
  filename = f;
  image = read_object(f);

   //does dsrv_obj exist?
   if(image == (OBJECT *) NULL)
   {
      printf("Illegal off file specification\n");
      exit(1);
   }

};

OFF_Drawable_Obj:: ~OFF_Drawable_Obj () {
//  delete filename;
//  delete image;
};

void OFF_Drawable_Obj:: ready_OFF_Obj() {
  ready_object_for_display(image);
};

void OFF_Drawable_Obj:: Draw() {

//  display_this_object(image);
```

APPENDIX G

NETWORK INTERFACE

```
//////////////////////////////////////////////////////////////////////
//////
//
//  This is a sample network class based on the VERN testbed
//  networking model produced in support of DARPA's SIMNET
//  by the Department of Computer Science at the University
//  of Central Florida, Orlando.
//
//  These classes are network stubs, only.  They have limited
//  functionality.  They are included to show the
architecture
//  of a network system and to outline the method for
networking
//  the DSRV simulator.
//
//////////////////////////////////////////////////////////////////////
//////

#include "Dynamic_Objs.H"
#include <math.h>

#ifndef _NETWORK_OBJECTS
#define _NETWORK_OBJECTS

//////////////////////////////////////////////////////////////////////
//////
//
//  Network_Player
//
//////////////////////////////////////////////////////////////////////
//////

class Network_Player {

  public:
```

```
    Network_Player();
    Network_Player(char *n)
;
    ~Network_Player();

    void set_error_tolerance(double, double, double);

    // methods which must be implemented for networking
interface
    void processMsg();

    // this method serves two funtions
    //  1) calculate new position and velocity based on
simulation time
    //  2) update state information of the player (internal
variables)
    void computeNextState(double, double, double);

  private:
    char *name;
    ThreeVector error_tolerance;

};


//////////////////////////////////// ////////////////////////
//////
//
//  Network_Ghost
//
////////////////////////////////////////////////////////////////
//////

class Network_Ghost {
  friend Network_Player;

  public:
    Network_Ghost();
    ~Network_Ghost();

    // method which must be implemented for networking
interface
```

```
    //   This is the dead reckoning method which will exist
in all
    //   machines showing this object
    //   It computes the Ghost's approximate state model
    void computeNextState(float);
    void update_state() {};

  protected:
    ThreeVector net_position;
    ThreeVector net_velocity;
};


#endif


//////////////////////////////////////////////////////////////
//
//  Network_Interface.C
//
//////////////////////////////////////////////////////////////


#include "Network_Interface.H"

//////////////////////////////////////////////////////////////
//////
//
//  Network_Player
//
//////////////////////////////////////////////////////////////
//////

Network_Player:: Network_Player() {};

Network_Player:: Network_Player (char *n) {
  name = n;
};

Network_Player:: ~Network_Player() {};

void Network_Player:: set_error_tolerance (double x, double
y, double z) {
```

```
  error_tolerance.x = x;
  error_tolerance.y = y;
  error_tolerance.z = z;
};

void Network_Player:: computeNextState (double x, double y,
double z) {

  // sample comparisons to determine when to update network
ghosts
  /*
  if (   fabs(x - Network_Ghost::net_position.x) >
error_tolerance.x
      || fabs(y - net_position.y) > error_tolerance.y
      || fabs(z - net_position.z) > error_tolerance.z) {

    send message to local and remote ghosts to update
position

  }
  */
};

//////////////////////////////////////////////////////////////
//////
//
//  Network_Ghost
//
//////////////////////////////////////////////////////////////
//////

Network_Ghost:: Network_Ghost() {};

Network_Ghost:: ~Network_Ghost() {};

void Network_Ghost:: computeNextState (float dt) {
    net_position.x += net_velocity.x * dt;
    net_position.y += net_velocity.y * dt;
    net_position.z += net_velocity.z * dt;

    // Use this state information for displaying the object
on
    // remote nodes.
```

```
        // The Network_Player is responsible for monitoring the
        // the ghost to determine when threshold errors in
        // position are reached.
};
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943	2
Commander Gary J. Hughes, USN Chairman, Code Hu Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
Dr. Robert B. McGhee Code Mz, Computer Science Department Naval Postgraduate School Monterey, CA 93943	4
Dr. David R, Pratt Code Pr Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
Lieutenant Commander S. N. Zehner 1212 Cogliandro Dr. Chesapeake, VA 23320	2
Lieutenant Commander Tony King Curricular Office Computer Science Department Naval Postgraduate School Monterey, CA 93943	1